Développement de la plate-forme expérimentale ParkView pour la reconstruction de l'environnement dynamique

Frederic Helin

Conservatoire National des Arts et Métiers

Centre régional Rhône-Alpes

Centre d'Enseignement de GRENOBLE

Mémoire présenté par

Frédéric Pierre Hélin

en vue d'obtenir

Le Diplôme d'Ingénieur C.N.A.M.
en Informatique

Développement de la plate-forme expérimentale ParkView pour la reconstruction de l'environnement dynamique

Soutenu le 1er juillet 2003

Jury

Président : Mme Véronique Donzeau-Gouge

Membres : M. Christian Carrez

M. Jacques Courtin

M. Thierry Fraichard

M. Jean-Pierre Giraudin

M. Christian Laugier

M. André Plisson

Conservatoire National des Arts et Métiers

Centre régional Rhône-Alpes

Centre d'Enseignement de GRENOBLE

Mémoire présenté par

Frédéric Pierre Hélin

en vue d'obtenir

Le Diplôme d'Ingénieur C.N.A.M.
en Informatique

Développement de la plate-forme expérimentale ParkView pour la reconstruction de l'environnement dynamique

---

Les travaux relatifs au présent mémoire ont été effectués au sein de *CyberMove*, projet de recherche commun entre l'*INRIA*, le *CNRS*, l'Institut National Polytechnique de Grenoble, et l'Université Joseph Fourier, sous la direction du Dr. Christian LAUGIER et du Dr. Thierry FRAICHARD.

Table des matières

Table des figures

Remerciements

Je tiens tout d'abord à remercier :
- Christian LAUGIER, directeur du projet *CyberMove,* l'un de mes professeurs de robotique, pour m'avoir proposé ce sujet et aidé à en définir les contours,
- Thierry FRAICHARD, chargé de recherche à l'*INRIA*, qui a suivi ce travail et qui m'a accordé sa confiance pour le poursuivre, une année encore, dans le cadre d'un programme de recherche commun entre l'*INRIA* et *le CNRS,*
- James L. CROWLEY, directeur du projet *Prima*, pour m'avoir toujours accordé du temps, soit directement, soit par l'intermédiaire des membres de son équipe,
- Véronique DONZEAU-GOUGE, professeur au *CNAM* à Paris, qui me fait l'honneur de présider le jury,
- Jean-Pierre GIRAUDIN, responsable pédagogique et scientifique du 3ème cycle *CNAM-Informatique* à Grenoble,
- Christian CARREZ, professeur au *CNAM* à Paris,
- André PLISSON, directeur du centre d'enseignement *CNAM* de Grenoble,
- Jacques COURTIN, professeur à l'université Pierre Mendès France, directeur de l'*IUT2* lorsque j'y faisais mes études.

Parmi les gens que j'ai côtoyés durant ce stage, je remercie, par ordre alphabétique :
- Gérard BAILLE, des moyens robotiques de l'*INRIA*,
- Yves BARD, des services généraux,
- Francis COLAS, doctorant au sein de l'équipe *CyberMove*,
- Frédéric LARGE, doctorant au sein de l'équipe *CyberMove*,
- Markus MICHAELIS, ingénieur au sein de l'équipe *Movi*,
- Matthieu PERSONNAZ, ingénieur au sein de l'équipe *Movi*,
- Justus PIATER, ingénieur au sein de l'équipe *Prima*,
- Cédric PRADALIER, doctorant au sein de l'équipe *CyberMove*,
- Stéphane RICHETTO, ingénieur au sein de l'équipe *Prima*,
- Pierre-Jean RIVIERE, ingénieur au sein de l'équipe *Prima*,
- Cyril ZORMAN, stagiaire de *DESS* au sein de l'équipe *CyberMove*, avec qui je me suis associé pour fonder la société *probeSys*[1].

Je remercie également les services du *CUEFA*[2], et plus particulièrement Annie MOUNET, pour avoir été toujours disponible tout au long de cette formation.

Enfin, une pensée pour Francine BENNY, professeur de communication et de management lors du *DEST* (cycle B du *CNAM*), qui savait nous faire trouver l'ambition nécessaire pour aborder ces études d'ingénieur et beaucoup d'autres choses.

[1] *probeSys SSLL, SARL Scop*, société de services en logiciel libre, http ://www.probesys.com .
[2] Centre Universitaire d'Education et de Formation des Adultes de Saint Martin d'Hères (38).

Chapitre 1

Introduction

Ce travail de mémoire s'inscrit dans le projet de recherche *CyberMove* de l'INRIA dirigé par le Dr. Christian LAUGIER. Ce projet s'intéresse à la modélisation, à l'autonomie de mouvement ainsi qu'aux interactions 3D en robotique. L'un des concepts visé par ce programme concerne "la route automatisée" du futur. L'équipe propose des éléments de réponse au problème de la sécurité routière, au débit des infrastructures, à la vitesse de déplacement en intégrant également les impacts sur l'environnement. Les exemples d'applications sont les trains de véhicules avec ou sans conducteur, l'automatisation de certains transports publics, la route automatisée pour les camions, l'autoroute à haut débit, etc.

Le projet de recherche CyberMove

CyberMove, anciennement *SHARP* [Sha01], est un projet de recherche commun entre l'*INRIA*, le *CNRS*, l'*INPG*, et l'*Université Joseph Fourier* de Grenoble depuis 1991 ; il est localisé à l'*INRIA Rhône-Alpes*, et appartient également au laboratoire *GRAVIR* (Lab. d'Informatique Graphique, Vision et Robotique) de la fédération Imag.

Le projet CyberMove centre son activité de recherche sur l'étude des problèmes liés à la modélisation et à la génération automatique du mouvement et des interactions physiques en robotique. Le terme *robotique* revêt ici un caractère particulier, dans le sens où il inclut à la fois des machines physiques (communément appelées *robots*) capables d'actions autonomes dans le monde réel, et des agents mobiles ou articulés (où *robots virtuels*) possédant des capacités de mouvements propres leurs permettant d'évoluer de manière autonome (ou semi-autonome) dans un monde virtuel possédant des lois physiques semblables à celles du monde réel.
CyberMove développe dans ce cadre :

1. une algorithmique pour la planification de mouvements avec prise en compte explicite de contraintes classiques de non collision, mais aussi de contraintes additionnelles provenant

de la nature physique du monde dans lequel les robots (réels ou virtuels) évoluent (e.g. contraintes cinématiques et dynamiques, incertitude) ;

2. une méthodologie pour le développement d'architectures décisionnelles pour le contrôle *intelligent* des mouvements, actions, et interactions de robots évoluant dans des environnements dynamiques peu ou pas connus a priori ;
3. des modèles et algorithmes pour gérer les interactions physiques et simuler la dynamique des corps complexes en mouvement et en interaction (déformations, collisions, forces...), en traitant de manière unifiée les mécanismes graphiques et haptiques d'interaction avec l'opérateur humain ;
4. des outils de modélisation et de calcul probabiliste pour la géométrie, afin de pouvoir traiter correctement (i.e. en exploitant le cadre formel de la théorie des probabilités et du calcul bayésien) les incertitudes et leurs impacts sur les problèmes inverses et les problèmes d'interprétation de données sensorielles que nous rencontrons (ce dernier sujet étant traité en collaboration étroite avec l'équipe Laplace du Laboratoire Leibniz de l'*IMAG*).

L'activité de recherche précédente est à la fois valorisée et fertilisée par des activités plus appliquées qui visent au développement de solutions à des problèmes industriels. Plusieurs prototypes de recherche et expérimentations réelles (e.g. sur des robots manipulateurs industriels, des véhicules autonomes, où des systèmes de vidéo professionnelle et de production d'images) sont ainsi réalisés en relation avec les moyens robotiques de l'*INRIA* Rhône-Alpes et des industriels ; certains de ces prototypes ont déjà donné lieu à des transferts de technologies (en CAO-Robotique et en vidéo professionnelle en particulier).

Les applications plus particulièrement visées par cette activité de recherche sont celles de la robotique non manufacturière (e.g. maintenance d'équipements ou intervention en milieu hostile ou lointain, robotique de service...), en mettant l'accent sur les domaines du transport et du médical ; l'autre secteur d'application concerné par nos travaux sur le mouvement dans le monde virtuel est celui de la réalité virtuelle et du multimédia, en mettant actuellement l'accent sur les domaines des effets spéciaux en vidéo et des outils de création de jeux vidéos.

L'autonomie de mouvement

L'équipe *CyberMove* a développé au cours des dernières années les bases d'une architecture de contrôle visant à doter un robot mobile de type voiture, tel que le *Cycab* de L'*INRIA* présenté figure 1.1, de la capacité de se déplacer de façon autonome. Les modes de conduite assistés par ordinateur permettent une simplification et une sécurisation de la conduite pouvant

aller jusqu'à une conduite totalement automatique sur des voies réservées. Ces recherches portent sur la planification des chemins et l'exécution de ceux-ci. Cette exécution ne consiste par seulement à suivre au mieux le chemin déterminé, elle doit en plus adapter le mouvement à la réalité perçue. Le travail présenté dans ce mémoire concerne spécifiquement cette perception de l'environnement.

FIG. 1.1 – Le robot mobile Cycab de l'INRIA

Objectif de ce travail

L'objectif de ce travail consiste à développer une plate-forme expérimentale destinée à la reconstruction de l'environnement dynamique. Cette reconstruction suppose d'être capable de suivre les déplacements de l'ensemble des éléments composant l'environnement. Ce suivi de cibles mobiles s'appuie sur l'analyse en continu des données provenant d'un ou plusieurs capteurs. Ce concept se concrétise sous forme d'un système à contraintes temps réel capable de réagir aux événements extérieurs et de les interpréter de manière "intelligente". Ainsi, nous verrons qu'il sera utilisé des caméras et un laser télémétrique à bord du véhicule. Le développement de cette plate-forme, nommée *ParkView*, est destinée à faciliter les expérimentations dans les domaines de la robotique mobile ainsi que ceux de la vision intelligente dans des images vidéo en temps réel.

Bien que ce travail ait été effectué dans le cadre de l'équipe *CyberMove*, il a été fait appel aux compétences des équipes de recherche *Prima* et *Movi* du laboratoire *IMAG-GRAVIR*, spécialisées dans la vision.

Les collaborations

le projet *Prima*

L'objectif du projet *Prima* [Pri01] est l'élaboration des fondements scientifiques des environnements *interactifs*. Un environnement *interactif* repose sur des capacités de perception, d'action et de communication. Un environnement est dit « perceptif » s'il est capable de maintenir un modèle de ses occupants et de leurs activités. Un environnement devient « actif » quand il est doté d'une capacité d'agir. Ces actions peuvent inclure de simples présentations d'information jusqu'à la capacité de gérer les communications acoustiques et visuelles et à transporter des documents et des matériels.

le projet *Movi*

Le projet *Movi* (modélisation, localisation, reconnaissance et interprétation en vision par ordinateur) vise à apporter des solutions générales et fondamentales permettant d'identifier, localiser, mesurer des formes vues à travers une ou plusieurs caméras, et de valider ces solutions à travers des applications diverses allant de la robotique jusqu'à la consultation de bases d'images. Les outils de base développés au sein du projet sont la géométrie de la perception tridimensionnelle, les invariants associés à cette géométrie, la mise en correspondance image - image ou image - modèle.

Organisation de ce mémoire

Dans le chapitre 2, nous définirons les objectifs de l'infrastructure *ParkView* et présenterons les solutions retenues pour les atteindre.

Dans le chapitre 3, nous nous intéresserons à la notion de carte de l'environnement dynamique. Nous proposerons une formalisation algébrique de la problématique et exposerons l'algorithme retenu pour *ParkView*.

Les chapitres 4 et 5 présenteront l'étude de l'infrastructure matérielle et les développements logiciels.

Enfin, nous présenterons dans le chapitre 6 deux applications utilisant notre infrastructure. Celles-ci concernent la mesure automatique de la vitesse sur route et l'évitement d'obstacles mobiles pour un robot de type *Cycab*.

Chapitre 2

ParkView

Ce chapitre s'intéresse dans une première partie à la définition des objectifs de *ParkView*, nom donné à la plate-forme expérimentale que nous avons installée sur le parking arrière de l'*INRIA Rhône-Alpes*. Dans une seconde partie, nous exposerons les solutions retenue pour la réalisation de *ParkView*.

2.1 Les objectifs de *ParkView*

L'objectif de *ParkView* est d'équiper le parking arrière de l'*INRIA* d'un système de perception dont le rôle principal est de construire un modèle de cet environnement incluant les objets fixes et mobiles ; on parlera alors de reconstruction de l'environnement dynamique.

Cette plate-forme expérimentale est destinée à faciliter les expérimentations dans les domaines de la recherche sur la vision et la robotique mobile. Devant être utilisée par plusieurs équipes de recherche, elle sera ouverte et extensible et capable d'offrir différents niveaux d'informations allant des signaux analogiques des capteurs jusqu'aux cartes de l'environnement dynamique.

Dans une première partie, nous définirons ce qu'est la reconstitution de l'environnement dynamique . Nous nous intéresserons ensuite aux notions d'extensibilité et d'ouverture de notre architecture. Enfin, nous nous intéresserons au traitement de l'information provenant de capteurs hétérogènes et à sa mise à disposition en temps réel.

2.1.1 La reconstruction de l'environnement dynamique

La reconstruction de l'environnement consiste à fournir une carte remise à jour périodiquement de l'ensemble des objets fixes et mobiles présents dans la zone de couverture de *ParkView*. Ceux-ci peuvent être aussi divers que les limites de la chaussée, des véhicules, des piétons ou encore des robots autonomes utilisateurs de notre infrastructure ou non. *ParkView*

n'a pas, dans un premier temps, pour but de renseigner sur leur nature. Elle devra toutefois être capable de renseigner sur certaines de leurs caractéristiques invariantes telle que leur dimension.

Les aspects statiques

ParkView est installé sur le parking de l'*INRIA Rhône-Alpes*. Il faudra prendre en compte les obstacles statiques dans cette zone de couverture. Ceux-ci pourront être des voitures garées sur le parking et donc immobiles ou des éléments du relief tel que des bordures, trottoirs, bornes, bâtiments, etc.

Les aspects dynamiques

Une carte de l'environnement dynamique devra renseigner sur la position des obstacles à un instant donné mais également sur leur vitesse et leur direction. Ces informations devront être suffisamment précises pour permettre à des robots mobiles d'anticiper les évolutions de leurs environnements. En ce sens, nous présenterons dans le chapitre 6.2 une expérimentation réalisée à l'aide de *ParkView,* où un robot mobile devra suivre au mieux un chemin planifié tout en évitant des obstacles mobiles.

2.1.2 Ouverture

En fonction des besoins des expérimentations, les différents niveaux de l'architecture *ParkView* devront être accessible. Ainsi, dans le cadre des travaux sur la robotique mobile, il est utilisé le plus haut niveau d'information, les cartes de l'environnement, sans s'intéresser, dans la mesure du possible, à la nature des capteurs qui ont servi à le constituer. Au contraire, les équipes travaillant sur la vision automatisée sont intéressées par avoir accès à des images brutes et parfois même directement aux signaux issus des différents capteurs. La figure 2.1 illustre la multiplicité d'accès aux différents niveaux d'informations.

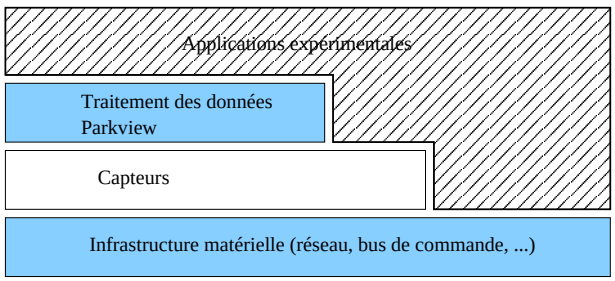


FIG. 2.1 – Les différents niveaux d'interface de *ParkView*

2.1.3 Extensibilité

Il doit être facile d'ajouter à *ParkView* des fonctionnalités et des matériels nouveaux. En ce sens, les choix que nous ferons lors de sa conception ne devront pas être limitatifs pour son extensibilité. Il est, par exemple, déjà prévu d'étendre *ParkView* sur une surface de parking beaucoup plus importante en lui adjoignant de nouveaux capteurs et leurs traitements.

2.1.4 Prise en compte de plusieurs capteurs

Notre architecture sera multi-capteurs. Cela est rendu nécessaire par la surface de parking à couvrir, mais aussi par la nature des informations demandées. Nous verrons que nous utilisons plusieurs caméras vidéo ainsi qu'un capteur de positionnement laser.

L'hétérogénéité des moyens de collecte de l'information doit-être transparente pour les dispositifs utilisant *ParkView*. Ce travail de reconstruction est donc aussi celui de la fusion de l'ensemble de ces données en une information globale homogène.

2.1.5 Mise à disposition de l'information

Pour que la carte de l'environnement soit suffisamment à jour, il faut prendre en compte la problématique de la transmission de l'information. Celle-ci s'entend, d'une part, entre les robots mobiles demandeurs d'informations sur l'environnement et l'infrastructure *ParkView* et, d'autre part, entre les éléments internes à l'architecture *ParkView*. A tous ces niveaux, il faut avoir à l'esprit l'aspect temps réel puisque les acteurs de l'environnement, qu'ils soient robots ou obstacles mobiles, évoluent sur le parking à des vitesses parfois relativement élevées.

2.1.6 La continuité de fonctionnement

L'application *ParkView* doit-être pensée comme devant, à terme, pouvoir fonctionner en continu. Ceci implique une grande fiabilité de fonctionnement puisque le système devra être "disponible" pour des dispositifs se présentant au fur et à mesure, sans connaître, a priori, le moment de la fin de l'expérimentation.

Cette capacité à pouvoir fonctionner pendant une durée indéfinie oblige à une utilisation rigoureuse des ressources, notamment dans les opérations d'allocation mémoire lors des phases de développement logiciel.

2.1.7 Choix technologiques

Les équipes devant se servir de *ParkView* ont la volonté de développer des technologies utilisables à court terme dans l'industrie. En ce sens, elles ne désirent pas les tester dans des

conditions idéales, mais, au contraire, souhaitent les éprouver en conditions réelles. Le choix des technologies sera donc orienté par des contraintes techniques mais également économiques. Il concernera le matériel : capteurs, stations de travail, mais aussi la partie logicielle dans la sélection et la mise en oeuvre des bibliothèques utilisées.

2.2 Solutions apportées

Dans cette partie, nous exposons les solutions apportées pour la mise en place de *ParkView*.

2.2.1 Une architecture en bus

Afin de répondre aux problématiques d'ouverture et de mise à disposition de l'information, nous avons opté pour une architecture en bus. Celle-ci permet à chaque équipe de venir connecter ses équipements d'acquisition et de traitement à n'importe quel niveau de l'infrastructure *ParkView*.

L'architecture en bus mise en place, présenté sur la figure 2.2, transporte les signaux vidéo et de synchronisation, la commande des capteurs et le réseau.

Le bus vidéo

Comme cela est décrit en 2.2.2, les capteurs utilisés sont pour l'essentiel des caméras vidéo analogiques. Le transport des images se fait donc par des signaux analogiques depuis les sites de prise de vue jusqu'aux stations d'acquisition. Afin de permettre l'utilisation de matériel d'acquisition différents, l'architecture en bus permet d'en connecter plusieurs sur un même capteur.

Nous verrons en 4.3.2 que l'infrastructure mise en place permet d'étendre facilement les capacités du bus vidéo par ajout de lignes supplémentaires.

Le bus de synchronisation

Le bus de synchronisation permet de synchroniser les caméras vidéo. Dans le cadre de la reconstruction de l'environnement en trois dimensions faite par l'équipe *Movi*, il est indispensable que les caméras utilisées acquièrent leurs images au même moment. Pour cela, elles sont équipées d'une entrée de synchronisation extérieure.

Nous avons mis en place un bus de synchronisation pour chaque caméra et non un bus commun. Cela permet de créer des groupes de synchronisation indépendants.

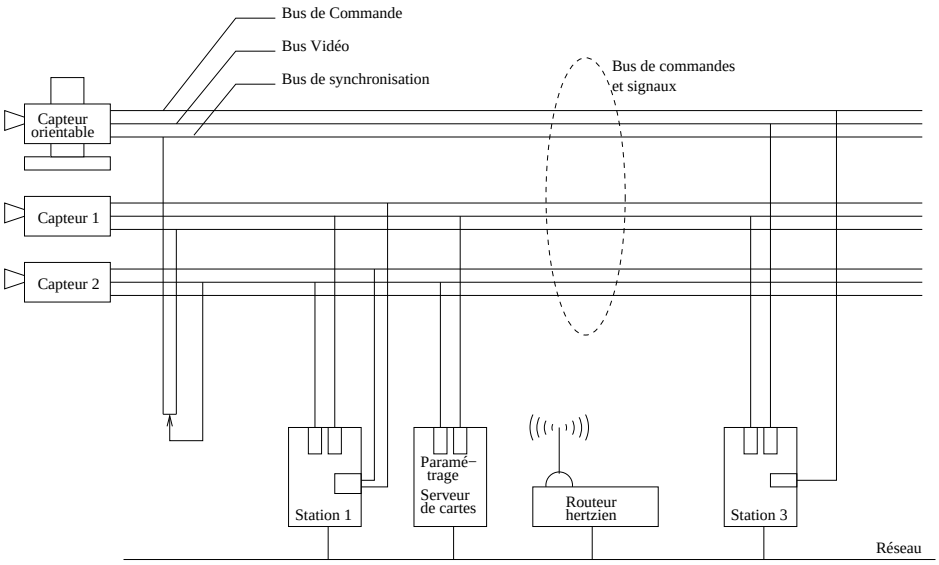


FIG. 2.2 – L'architecture en bus de *ParkView*

Le bus de commande

L'infrastructure de *ParkView* étant extérieur, dans la mesure du possible, nous choisissons d'utiliser des capteurs paramétrables à distance. Nous avons mis en place un bus de commande pour chaque capteur afin de faciliter les connexions avec les cartes de commande des différentes équipes.

2.2.2 Le matériel utilisé

L'application *ParkView*, bien qu'étant un projet développé par l'équipe *CyberMove*, s'est adjoint les compétences des équipes travaillant sur la vision *Prima* et *Movi*. Ces équipes ont depuis longtemps la connaissance de matériels avec lesquels elles travaillent. Ces projets de recherche ayant la volonté de proposer des solutions directement utilisables dans l'industrie, ils travaillent souvent avec des matériels de grande série. Dans notre cas, outre le gain de temps que nous apporte l'intégration de leur technologie, il existe un aspect économique certain à utiliser ces matériels. Les choix sont également motivés par la volonté de créer une infrastructure facilement extensible à des coûts raisonnables.

2.2.3 Modularité

La volonté de l'équipe *CyberMove* est de pouvoir, à terme, disposer d'une infrastructure capable de suivre l'ensemble des déplacements sur le parking de l'*INRIA Rhône-Alpes*. Cette architecture répartie sépare les tâches de paramétrage, d'acquisition / traitement et de mise

à disposition de l'information. Cette dernière fonctionnalité est confiée à un serveur de carte interrogé par les robots mobiles.

2.2.4 Le serveur de carte dynamique

Afin de mettre à disposition les cartes de l'environnement, nous avons mis en place une architecture client/serveur. Celle-ci s'articule autour d'un serveur maintenant à jour une carte de l'environnement dynamique consulté par les robots mobiles. La communication est assurée par un réseau *LAN* hertzien dont le routeur est présenté sur le schéma de la figure 2.2. Dans cette version de *ParkView*, nous verrons que l'acquisition des données, les traitements et le serveur sont réalisés par la même machine. Nous verrons dans la partie 5.5 que nous utilisons pour cela des composants logiciels asynchrones permettant d'avoir des temps de traitements différents sans pénaliser l'ensemble.

Chapitre 3

Construction de la carte de l'environnement dynamique

Après avoir défini en 2.1.1 les caractéristiques d'une carte de l'environnement dynamique, nous détaillerons dans ce chapitre les étapes nécessaires à sa construction.

Dans une première partie, nous présenterons le *tracker* de l'équipe *Prima* qui est le dispositif de suivi de cible que nous avons utilisé. Nous définirons le référentiel *ParkView* dans lequel seront exprimées les données de la carte. Nous nous intéresserons ensuite aux transformations des données issues du *tracker* vers le repère de référence puis, nous détaillerons la fusion de ces données, issues de plusieurs *trackers,* permettant la construction de la carte. Dans une dernière partie, nous nous intéresserons aux aspects temps réel du transport des informations issues des *trackers* et des cartes sur le réseau. La réalisation logicielle de cette reconstruction sera décrite dans la chapitre 5.

3.1 Le *Tracker*, le module de suivi de cible *Prima*

La *Tracker* développé par l'équipe *Prima* est un système capable de suivre en temps réel plusieurs cibles dans un flux d'images vidéo. Il renvoie la position de ces cibles, leurs dimensions sous forme de paramètres d'une ellipse englobante et leur vitesse, voir figure 3.1.

Ces informations sont données dans le système de coordonnées de la caméra et de la carte d'acquisition. Ainsi, lorsque plusieurs systèmes caméra-*tracker* observent un même objet, les informations qu'ils retournent ne sont pas données dans le même référentiel. Nous avons donc défini un repère commun, appelé référentiel *ParkView*, afin de pouvoir représenter l'ensemble de ces information dans un même espace. Le chapitre 5, consacré au développement logiciel, reviendra plus en détail sur l'architecture et l'implémentation du *tracker Prima*.

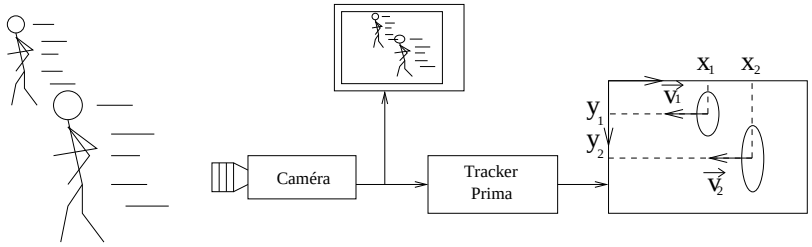


FIG. 3.1 – Le suivi de cible *Prima*

3.2 Le référentiel *ParkView*

Le référentiel *ParkView* est l'espace de représentation commun des données. C'est dans ce repère que sont exprimées les coordonnées des objets de la carte de l'environnement dynamique. Il est donc nécessaire de convertir les données issues de nos capteurs vers le repère de référence tel qu'illustré par la figure 3.2. Nous verrons que c'est une fois ces conversions faites qu'il sera possible de fusionner les observations provenant des différentes sources. Pour cela, il est nécessaire de déterminer la fonction inverse du modèle capteur. Dans la partie suivante, nous considérerons le capteur comme étant le couple caméra-*tracker*.

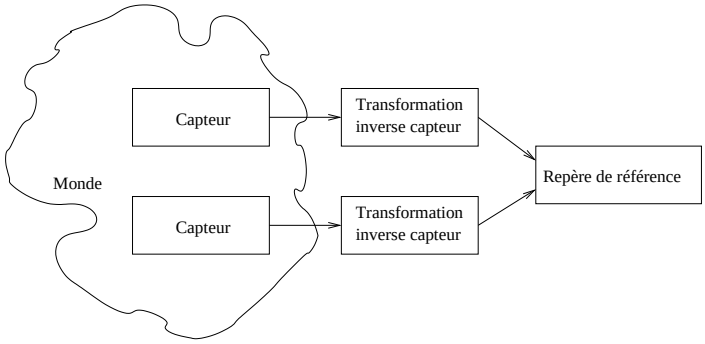


FIG. 3.2 – Transformation vers le repère de référence

3.3 La fonction inverse capteur

3.3.1 Formalisation

La fonction de transformation du modèle capteur a pour résultat une observation dans un référentiel propre au capteur, R_c, d'un phénomène dans le référentiel du monde réel, R_m. Cette transformation peut-être exprimée sous la forme :

$$O = f(\Phi)$$

où O est l'observation par le capteur d'un ensemble de phénomènes Φ. Cette fonction de transformation est assurée par le capteur.

Dans notre cas, il nous faut déterminer une fonction inverse nous permettant de passer d'un ensemble d'observations, O', dans R_c, vers un ensemble de phénomènes perçus, Φ', dans le repère de référence R_r.

Soit g cette fonction, nous écrivons la relation :

$$\Phi' = g(O').$$

Avec précaution, en excluant le fait que certains phénomènes puissent ne pas être perçus par notre capteur, il est possible d'assimiler Φ à Φ'. Cela revient donc à assimiler R_r à R_m et à identifier g à f^{-1}.

3.3.2 Composition de la fonction inverse capteur

La composition de la fonction inverse capteur demande de connaître, d'une part, le repère dans lequel le capteur travaille et à déterminer, d'autre part, la transformation depuis celui-ci vers le repère de référence.

Le repère dans lequel travaille notre capteur est celui de la caméra. Les données issues de nos caméras sont converties par des cartes d'acquisition en données numériques. Les images sont alors modélisées sous la forme d'une matrice de points telle que représentée sur la figure 3.3.

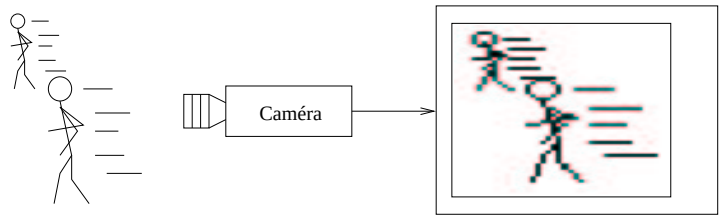


FIG. 3.3 – L'image vidéo, une matrice de points

Les caméras que nous utilisons sont équipées d'objectifs grand angle dont les caractéristiques sont déterminées dans le chapitre 4. Nous savons que ce type d'objectif engendre un phénomène de distorsion optique. Elle doit être corrigée avant d'opérer les projections depuis le repère de capteur vers celui de référence.

Dans la partie suivante, nous nous pencherons sur la distorsion optique. Nous définirons ensuite la fonction de projection.

La correction de la distorsion

Modèle sténopé ou trou d'épingle Le modèle simplifié généralement utilisé pour représenter une caméra est le modèle sténopé[1] ou trou d'épingle [Cro02]. Nous utilisons des termes dérivés de l'optique pour décrire ce modèle géométrique.

Ce modèle est formé d'un centre optique C, le trou d'épingle, et d'un plan image P. Tous les rayons lumineux issus de l'objet observé dans l'espace passent par le centre optique et se propagent en ligne droite. On appelle :

- **axe optique** la perpendiculaire menée du centre optique sur le plan image,
- **point central** l'intersection de l'axe optique avec le plan image,
- **distance focale** la distance entre le centre optique et le point central.

La projection d'un point M de l'espace sur le plan image est centrale ou perspective. Sa trace I_m sur l'image est l'intersection de la droite (M, C) passant par le point de l'espace et le centre optique C de l'appareil de prise de vues, avec le plan image P. La figure 3.4 présente le modèle géométrique sténopé, et le référentiel qui lui est associé. Il faut préciser qu'il est nécessaire d'effectuer une transformation entre le repère de caméra (X, Y) et le système de coordonnées du capteur (I, J).

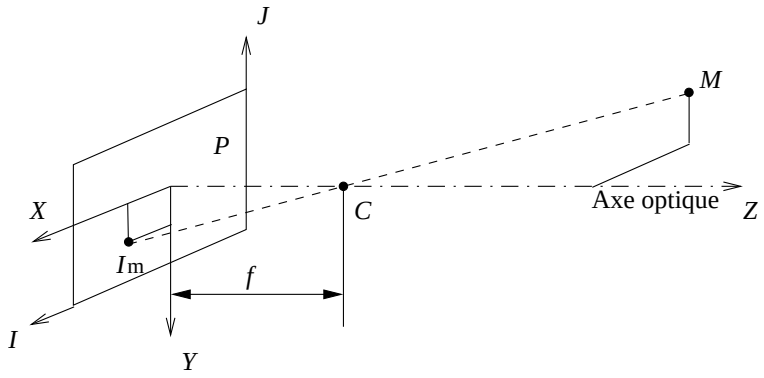


FIG. 3.4 – *Le modèle sténopé*

La distorsion optique Le modèle sténopé s'applique aux dispositifs équipés de lentilles minces, souvent de focale importante, n'engendrant pas de déformation optique. Les camé-

[1] (du grec sténos « étroit », et opé, « ouverture »). En photographie, dispositif d'une chambre noire dans laquelle la lumière pénètre non à travers un objectif, mais par un très petit trou percé dans la paroi. L'image obtenue est exempte de toute aberration.

ras que nous utilisons pour *ParkView* ne respectent par ce modèle. En effet, étant équipées d'objectifs grand-angle, il résulte une distorsion optique, voir figure 3.5. Il est important de corriger cette distortion [PWH97] [TYO02] [Ste97] [Mar95] avant d'appliquer les transformations homographiques que nous verrons par la suite.

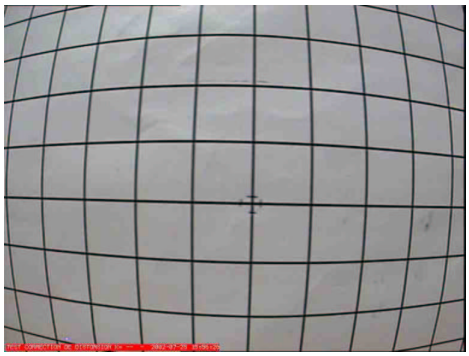


FIG. 3.5 – *Effets de la distorsion sur une grille de test orthogonale*

La distortion optique d'un système peut se définir en disant que les angles d'incidence des rayons lumieux ne sont pas conservés lorsqu'ils passent par son centre optique, voir figure 3.6.

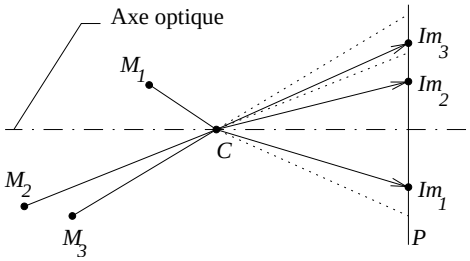


FIG. 3.6 – *Incidence des rayons passant par le centre optique*

Cette aberration est radiale autour le l'axe optique. Dans la figure 3.7, nous représentons les courbes de niveau de distorsion en tenant compte des erreurs d'appariement mécaniques entre le système optique et le capteur de la caméra. Nous distinguons pour cela le centre de l'image, C_i, du point central, C_{op}. Les points I_{m_i}sont les images des points M_i par la projection de centre C (modèle sténopé). Les points I'_{m_i}sont les images des même point M_i à travers un système ayant une distorsion optique.

La majorité des publications parues sur ce sujet admettent, pour leur modèle, que l'axe optique est parfaitement perpendiculaire au plan de projection P. Dans notre cas, une non perpendicularité à ce niveau entraîne une distorsion perspective qui est corrigée par les transformations homographiques ultérieures. Nous considérons donc également que l'axe optique et perpendiculaire à P.

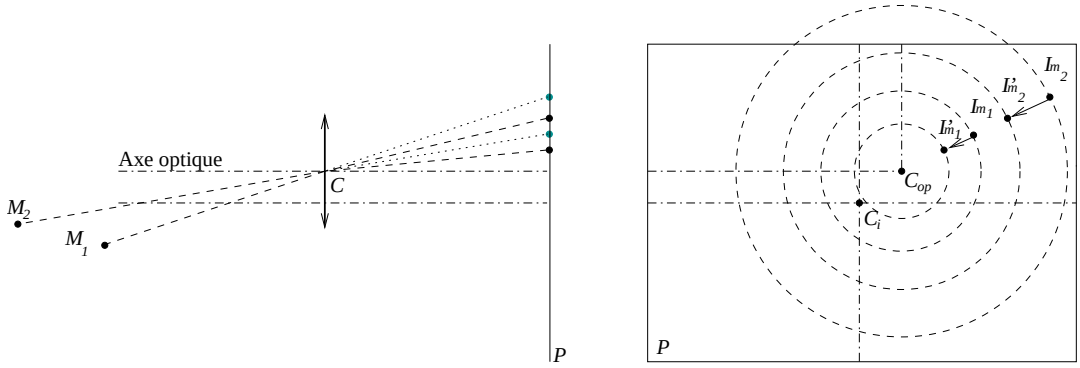


FIG. 3.7 – Projections autour de l'axe optique

Correction de la distorsion Le but de la correction de la distorsion est de recréer, à partir d'une image déformée, une image perspective telle que nous l'obtiendrions si nous disposions d'une caméra respectant le modèle sténopé. Dans cette partie, nous nous intéresserons donc à la relation qui existe entre la projection effectuée par la caméra et le modèle sténopé.

La projection géométrique de la plupart des caméras peut être modélisée comme la projection perspective du monde en trois dimensions sur une sphère (appelée aussi la sphère de vision), suivie par une projection π, depuis la sphère sur le plan image [Mar95], voir figure 3.8.

Pour la plupart des caméras du commerce, π est symétrique autour de l'axe optique et parallèle à celui-ci. Ainsi, dans le cas d'un point M projeté sur la sphère de vision puis sur le plan image P, représenté en coordonnées polaires, ces projections successives ne concerneront que la distance angulaire α depuis l'axe optique et préservera l'angle β autour de l'axe optique.

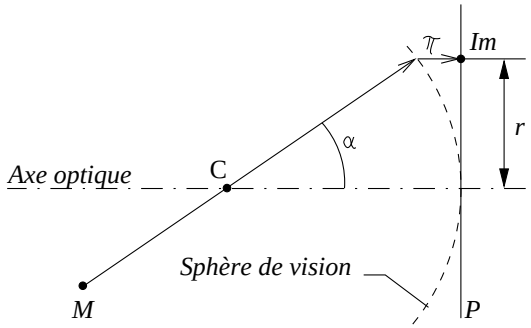


FIG. 3.8 – *Sphère de vision*

Le modèle géométrique En optique, lors de la conception de lentilles, cinq fonctions de projection radiale sont habituellement utilisées pour passer de la distance angulaire α au rayon $r(\alpha)$ depuis C. :

- perspective : $r(\alpha) = k.\tan(\alpha)$
- stéréographique : $r(\alpha) = k.\tan(\frac{\alpha}{2})$
- équidistante : $r(\alpha) = k.\alpha$
- angle equi-solide : $r(\alpha) = k.\sin(\frac{\alpha}{2})$
- sinus : $r(\alpha) = k.\sin(\alpha)$

Il est possible de faire le parallèle entre ces différentes projections et la géométrie des systèmes optiques. Ainsi, dans le cas de la projection perspective, le coefficient k peut être assimilé à la longueur focale f.

Le modèle polynomial D'une autre manière, il est possible d'utiliser une fonction polynomiale pour modéliser le rapport des distances r_d et r_c, respectivement, la distance entre C_{op} et le point de l'image déformée, I_m, et celle entre C_{op} et le point corrigé, I'_m. Cette relation introduit n coefficients $k_1...k_n$ pour quantifier la distorsion et est de la forme [TYO02] :

$$r_c = r_d(1 + k_1 r_d^2 + k_2 r_d^4 + k_2 r_d^6 + ... + k_n r_d^{2n}). \tag{3.1}$$

En prenant $n = 2$, comme c'est souvent le cas, les coordonnées du point p_d, provenant de l'image déformée et p_c, le point image corrigé, on a alors :

$$p_c = \begin{pmatrix} x_c \\ y_c \end{pmatrix} = \begin{pmatrix} \frac{x_d - c_x}{s_x}(1 + k_1 R^2 + k_2 R^4) + c_x \\ (y_d - c_y)(1 + k_1 R^2 + k_2 R^4) + c_y \end{pmatrix} \tag{3.2}$$

où $(c_x, c_y)^T$ sont les coordonnées du centre de distorsion, $R = \sqrt{((x_d - c_x)/s_x)^2 + (y_d - c_y)^2}$ et s_x, le ratio d'aspect des pixels définissant le rapport $\frac{largeur}{hauteur}$.

Pour *ParkView*, nous utilisons le modèle polynômial dont les effets sont présentés en annexe C. En effet, les constructeurs ne donnent pas le modèle de projection utilisé par leurs objectifs. Il est donc plus précis d'approcher le rapport des distances par détermination des coefficients k_i. Pour cela, il faut pouvoir mesurer la distorsion.

Mesure de la distorsion De nombreux travaux ont porté sur la mesure de la distorsion et plus généralement de la calibration de caméras. Les techniques employées reposent essentiellement sur deux méthodes : la calibration par l'utilisation de mires de test dont les caractéristiques sont parfaitement connues (voir figure 3.9) et la prise en compte de particularités connues dans le "paysage" (arrêtes de constructions parallèles, équidistances, angles droits connus, plans perpendiculaires, etc...) [WSB02] (figure 3.10).

La reconnaissance de ces points particuliers se fait soit par l'intervention d'un opérateur qui les pointe directement sur l'image, soit par des mécanismes automatiques de reconnaissance des formes [Dou00].

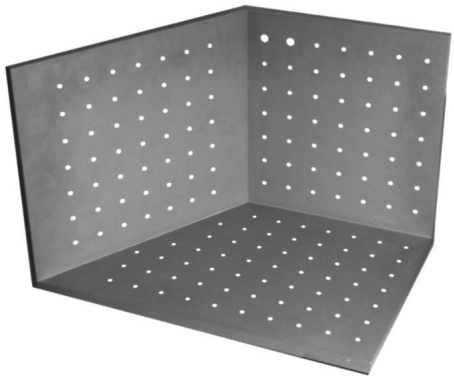


FIG. 3.9 – *Mire de calibration 3D*

FIG. 3.10 – *Mise en évidence de propriétés caractéristiques d'une scène [WSB02]*

Dans le cas de *ParkView*, nous utilisons la bibliothèque de calcul *Gandalf* [McL99] en lui fournissant des points supposés être sur des droites. Celle-ci nous retourne alors les coefficients $k_1$et k_2, et les coordonnées du centre optique dans l'image en effectuant une résolution par la technique des moindres carrés. Il suffit alors d'appliquer l'équation (3.2) pour obtenir les coordonnées corrigées.

Après correction, les paramètres des cibles renvoyées par le *tracker* sont exprimés dans un plan euclidien. Nous pouvons alors effectuer la projection depuis ce plan vers le plan de référence. Ceci est réalisé par une projection centrale appelée transformation homographique.

Transformation homographique

La passage du système de coordonnées du repère d'un capteur caméra vers le repère de référence *ParkView*, parallèle au plan du parking, est réalisé par une projection centrale passant par le centre optique, ou point nodal, de nos caméras, voir figure 3.11. La projection centrale ne conserve en général pas les milieux et par suite ne conserve ni le parallélisme ni les rapports de distance. C'est une transformation homographique non affine. Les transformations homographiques regroupent les translations, les homothéties, les symétries, les similitudes, les inversions, la perspective et l'homologie. Dans notre cas, nous ne chercherons pas à décomposer les homographies utilisées en transformations élémentaires mais les considérerons globalement.

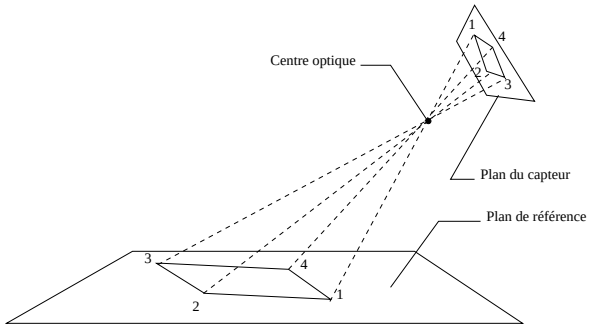


FIG. 3.11 – Projection centrale

Formalisation Soit les points P et P' appartenant à deux plans, nous définissons les relations H et H^{-1} telles que :

$$P'_h = H.P_h,$$

$$P_h = H^{-1}.P'_h,$$

où H est une matrice $H_{3\times 3}$ modélisant ici l'homographie réalisée par le système caméra entre le plan du parking et celui du capteur *CCD*. $P_h = (x_h, y_h, z_h)^T$ et $P'_h = (x'_h, y'_h, z'_h)^T$ sont la représentation de $P = (x, y)^T$ et $P' = (x', y')^T$ sous forme de vecteurs de coordonnées homogènes. Les relations entre les coordonnées homogènes et les coordonnées sont proportionnelles à z et z', soit :

$$x = \frac{x_h}{z}, \quad y = \frac{y_h}{z},$$

et de la même façon :

$$x' = \frac{x'_h}{z'}, \quad y = \frac{y'_h}{z'}.$$

A partir des coordonnées des cibles fournies par le *tracker* dans le repère de la caméra, il est possible de déterminer leurs coordonnées dans le repère du parking en appliquant H^{-1}.

Détermination des coefficients de l'homographie La détermination des coefficients h_{ij} de $H^{-1}_{3\times 3}$ revient à résoudre un système d'équations linéaires de la forme :

$$\begin{cases} P'_1 = \lambda.H^{-1}.P_1 \\ P'_2 = \lambda.H^{-1}.P_2 \\ \quad ... \\ P'_n = \lambda.H^{-1}.P_n \end{cases}.$$

La présence de λ permet de simplifier notre système sans influencer les résultats en coordonnées dans le plan. En effet, en prenant $\lambda = \frac{1}{h_{33}}$, nous pouvons considérer :

$$M = \lambda.H^{-1} = \begin{bmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ m_{31} & m_{32} & 1 \end{bmatrix}.$$

Avec huit coefficients, la résolution de ce système nécessite donc de connaître les coordonnées de quatre points et de leur image. Soit, pour un point :

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ m_{31} & m_{32} & 1 \end{bmatrix} \times \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} xm_{11} + ym_{12} + m_{13} \\ xm_{21} + ym_{22} + m_{23} \\ m_{31} + m_{32} + 1 \end{bmatrix}$$

$$\Leftrightarrow \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} \frac{x'm_{11}+y'm_{12}+m_{13}}{x'm_{31}+y'm_{32}+1} \\ \\ \frac{x'm_{21}+y'm_{22}+m_{23}}{x'm_{31}+y'm_{32}+1} \end{bmatrix} \quad avec \quad m_{31}+m_{32}+1 \neq 0$$

$$\Leftrightarrow \begin{cases} xm_{11}+ym_{12}+m_{13}-x'xm_{31}-x'ym_{32}=x' \\ xm_{21}+ym_{22}+m_{23}-y'xm_{31}-y'ym_{32}=y' \end{cases}$$

Il est possible le représenter ce système sous la forme $A.X = B$:

$$\begin{bmatrix} x' & y' & 1 & 0 & 0 & 0 & -xx' & -xy' \\ 0 & 0 & 0 & x' & y' & 1 & -yx' & -yy' \end{bmatrix} \times \begin{bmatrix} m_{11} \\ m_{12} \\ \vdots \\ m_{31} \\ m_{32} \end{bmatrix} = \begin{bmatrix} x \\ y \end{bmatrix}$$

Le système global avec les quatre couples de points s'écrira alors $A_{8\times 8}.X_{8\times 1} = B_{8\times 1}$:

$$\begin{bmatrix} x_1 & y_1 & 1 & 0 & 0 & 0 & -x_1x'_1 & -x_1y'_1 \\ 0 & 0 & 0 & x_1 & y_1 & 1 & -y_1x'_1 & -y_1y'_1 \\ x_2 & y_2 & 1 & 0 & 0 & 0 & -x_2x'_2 & -x_2y'_2 \\ 0 & 0 & 0 & x_2 & y_2 & 1 & -y_2x'_2 & -y_2y'_2 \\ x_3 & y_3 & 1 & 0 & 0 & 0 & -x_3x'_3 & -x_3y'_3 \\ 0 & 0 & 0 & x_3 & y_3 & 1 & -y_3x'_3 & -y_3y'_3 \\ x_4 & y_4 & 1 & 0 & 0 & 0 & -x_4x'_4 & -x_4y'_4 \\ 0 & 0 & 0 & x_4 & y_4 & 1 & -y_4x'_4 & -y_4y'_4 \end{bmatrix} \times \begin{bmatrix} m_{11} \\ m_{12} \\ \vdots \\ m_{31} \\ m_{32} \end{bmatrix} = \begin{bmatrix} x_1 \\ y_1 \\ \vdots \\ x_4 \\ y_4 \end{bmatrix}.$$

Nous connaissons A et B, la résolution consiste donc à inverser A, soit :

$$X = A^{-1} \times B.$$

La figure 3.12 illustre une transformation homographique pour les correspondances de point suivantes : $(10, 10) \rightarrow (30, 5)$, $(10, 20) \rightarrow (30, 17)$, $(25, 20) \rightarrow (40, 20)$, $(25, 10) \rightarrow (45, 10)$.

On remarquera que pour l'exemple nous effectuons ici une projection du plan dans lui-même.

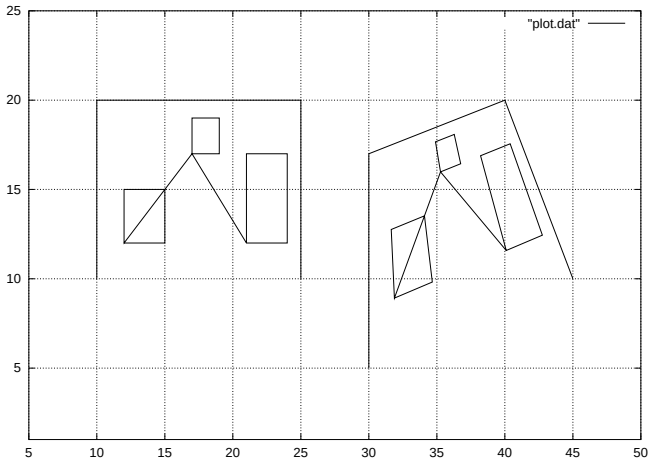


FIG. 3.12 – Exemple de transformation homographique

3.4 Construction de la carte de l'environnement

Dans les parties précédentes, nous avons défini la fonction inverse-capteur. Celle-ci nous permet de pouvoir exprimer les données issues de nos capteurs dans le repère de référence. Néanmoins, à ce stade, un même objet de notre environnement sera décrit autant de fois qu'il y aura de capteur le percevant. Nous devons donc effectuer une fusion de ces données capteurs afin de garantir l'unicité de l'information. La figure 3.13 présente le schéma général de l'application de reconstruction de l'environnement dynamique.

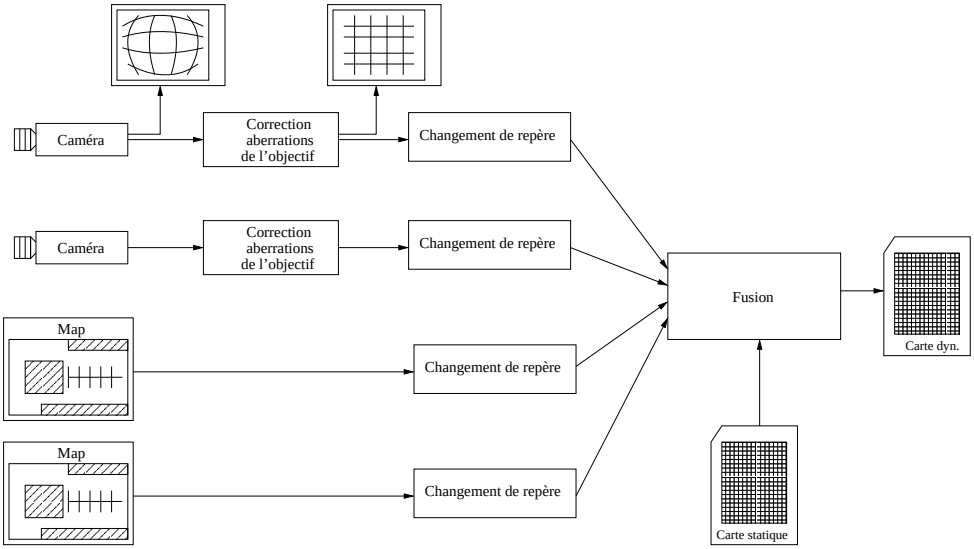


FIG. 3.13 – Schéma de l'application de reconstruction de l'environnement dynamique

Les cartes sont considérées ici de la même manière que les plans vidéo corrigés. Elles doivent nous permettre, à terme, de pointer directement les obstacles fixes de l'environnement à l'aide de la souris.

La partie suivante propose un formalisation algébrique de la fusion des données. Nous présenterons ensuite l'algorithme utilisé.

3.4.1 Perception multi-capteurs

Cette partie reprend la notation introduite en 3.3. Nous appellerons "cibles" ce que perçoivent les capteurs et "objets" l'interprétation qui en sera faite.

Nous définirons ici Φ' comme étant le phénomène global à percevoir sur le parking. Un capteur c_i n'assure qu'une observation partielle O_i de Φ'. Pour chaque capteur c_i, nous avons donc la connaissance Φ_i dans R_r avec :

$$\Phi_i \subset \Phi'.$$

Même si on cherche à l'éviter, il se peut que des cibles ne soient momentanément perçues par aucun capteur, voir figure 3.14. Ceci peut-être du à un mauvais placement de nos capteurs ou à une perte de cible du *tracker*.

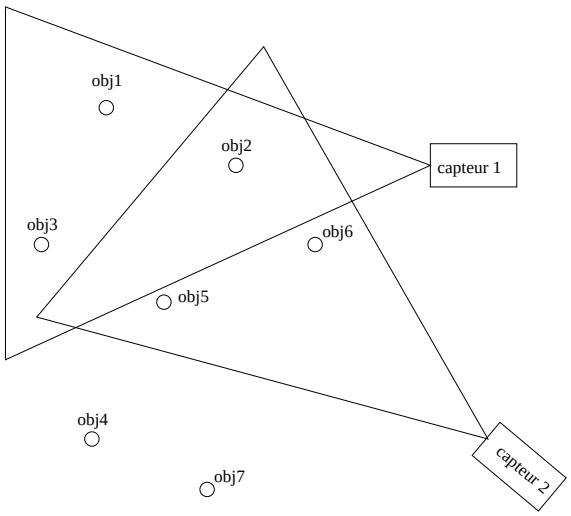


FIG. 3.14 – Cibles non perçues

Nous n'utiliserons donc pas l'égalité pour dire que :

$$\bigcup_{i \in I} \Phi_i \subset \Phi'. \tag{3.3}$$

Nous pouvons introduire l'égalité dans l'équation (3.3) en tenant compte de l'ensemble des phénomènes liés aux cibles non perçues Φ_{np} :

$$(\bigcup_{i \in I} \Phi_i) \cup \Phi_{np} = \Phi'.$$

Appariement des observations et création des objets associés

La multiplicité des capteurs ainsi que leur hétérogénéité ne devant pas transparaître pour les dispositifs utilisant l'application, il nous faut donc pouvoir reconnaître les objets mobiles perçus par plusieurs capteurs, afin de garantir l'unicité de l'information. Ici, nous nous intéresserons au cas où une ou plusieurs cibles sont perçues par deux capteurs. Nous définissons l'ensemble $\Phi_{com_{i,j}}$comme étant le phénomène commun à $\Phi_i$et Φ_j. Soit :

$$\forall i, j \in I, J, \Phi_{\mathrm{com_{i,j}}} = \Phi_i \cap \Phi_j.$$

Enumérer l'ensemble Φ_{com} revient à définir une méthode d'appariement entre deux groupes de cibles. Les capteurs utilisés nous renseignent sur la position des cibles suivies mais également, pour certains, sur leur vitesse, la direction de leur déplacement, ainsi que sur leur dimension perçue. L'ensemble des données pour une cible suivie pourra être formalisé comme un vecteur de paramètres P. Il nous faut maintenant savoir si deux vecteurs de données P_1 et P_2 peuvent être considérés comme décrivant un même objet. Pour cela, nous allons définir une fonction de distance.

Distance entre les cibles Nous considérons ici que nous avons déjà appliqué la fonction inverse au modèle capteur et que les données des vecteurs paramètre P_1 et P_2 sont exprimées dans le repère de référence commun. Nous définissons le scalaire d comme étant la distance entre P_1 et P_2 :

$$d = dist(P_1, P_2).$$

Nous avons le choix quant à la définition de $dist$. Dans notre cas, ce sera la somme pondérée de distance propre aux paramètres de même nature, soit, pour n paramètres :

$$dist(P_1, P_2) = \alpha_1.dist_1(p_{1_1}, p_{1_2}) + ... + \alpha_n.dist_n(p_{n_1}, p_{n_2}).$$

Si nous prenons l'exemple de la connaissance de la position et de la vitesse, P sera de la forme $\begin{pmatrix} x, & y, & v \end{pmatrix}^T$. Nous définirons $dist$ comme la somme pondérée de la distance euclidienne des positions et de la valeur absolue de la différence des vitesses (figure 3.15), soit :

$$dist(P_1, P_2) = \alpha_1.\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2} + \alpha_2.|v_2 - v_1|.$$

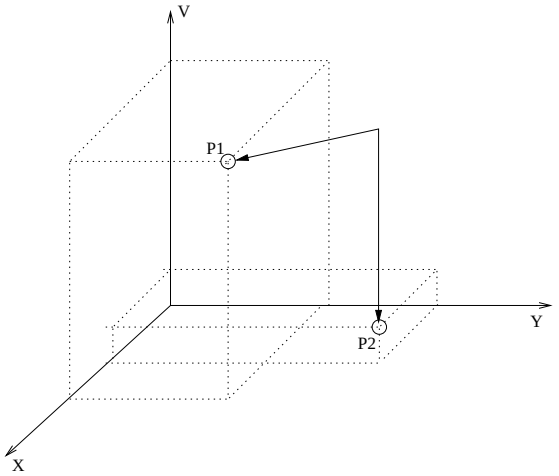


FIG. 3.15 – Distance entre deux points dans l'espace des paramètres x,y,v

La fonction *dist* pourra également prendre en compte la distance temporelle entre deux mesures dans le cas de l'utilisation de capteurs non synchronisés.

Fusion de paramètre

La fonction de distance nous renvoie le scalaire d. Il nous appartient de prendre une décision de mise en correspondance à partir de cette valeur, soit, que plusieurs cibles décrivent le même objet réel. Nous définissons un seuil, s, en-dessous duquel la mise en correspondance sera effectuée. Ce sera l'un des paramètres de notre application ; il sera redéfini à chaque modification de *dist*.

Lorsque $d < s$, il nous faut alors déterminer $\phi_{f_{1,2}}$ comme étant la perception globale de ϕ_1 et ϕ_2. Le vecteur $P_{f_{1,2}}$ est alors le vecteur paramètre de l'objet issu des deux observations de paramètres P_1 et P_2. Nous définissons pour cela la fonction de fusion fus :

$$P_{f_{1,2}} = fus(P_1, P_2).$$

Dans notre cas, lorsque $d < s$, seul $P_{f_{1,2}}$ sera connu des dispositifs connectés à *ParkView* et P_1 et P_2 seront cachés. Ceci est logique car, dans la réalité, seul un objet existe. Cela nous garantit l'unicité des données, c'est-à-dire, qu'un même objet ne sera décrit qu'une seule fois.

3.4.2 Algorithme de fusion proposé

On définit la notation suivante :

- O - L'ensemble des objets déjà connus dans le système. C'est cet ensemble qui modélise l'environnement connu par *ParkView* à un instant et qui est fourni aux dispositifs s'y connectant.
- o - La notation pour un objet particulier de l'ensemble O
- T_1, T_2 - L'ensemble des cibles suivies par les deux *trackers*.
- c, c_1, c_2 - Notations pour des cibles particulières.

Traitement des objets o, $o \in O$, connus :

for all $o \in O$ **do**

Vérifie l'existence de $o.c_1$ et $o.c_2$ dans T_1, T_2

if $o.c_1 \in T_1$ and $o.c_2 \in T_2$ **then**

– On vérifie la cohérence de l'objet o

if $dist(o.c_1, o.c_2) > s$ **then**

– S'il n'est pas cohérent, on enlève le lien entre l'objet et la cible la plus éloignée de $o.p$, nous verrons que $o.p = fus(o_{t-1}.c_1, o_{t-1}.c_2)$

if $dist(o.p, o.c1) < dist(o.p, o.c2)$ **then**

On enlève $o.c_2$ à o ($o.c_2 = NULL$),

On enlève $o.c_1$ à T_1

else

On enlève $o.c_1$ à o ($o.c_1 = NULL$),

On enlève $o.c_2$ à T_2

end if

else

o n'est pas modifié, on enlève $o.c_1$ et $o.c2$ à T_1 et T_2

end if

else if $o.c_1 \in T_1$ and $o.c_2 \notin T_2$ **then**

On enlève la référence périmée $o.c_2$ à o ($o.c_2 = NULL$)

On enlève $o.c_1$ à T_1

else if $o.c_1 \notin T_1$ and $o.c_2 \in T_2$ **then**

On enlève la référence périmée $o.c_1$ à o ($o.c_1 = NULL$)

On enlève $o.c_2$ à T_2

else

Suppression de o à O.
end if
– Vérifie l'entrée d'une nouvelle cible dans l'objet traité
if $o.c_1 == NULL$ **then**
for all $c \in T_1$ **do**
if $dist(c, o.c_2) < s$ **then**
$o.c_1 = c$
On enlève c à T_1
end if
end for
end if
if $o.c_2 == NULL$ **then**
for all $c \in T_2$ **do**
if $dist(c, o.c_1) < s$ **then**
$o.c_2 = c$
On enlève c à T_2
end if
end for
end if
end for

Traitement des nouvelles cibles, création des objets :

for all $c_1 \in T_1$ **do**
Création de o avec $o.c_1 = c$
On enlève c_1 à T_1
for all $c_2 \in T_2$ **do**
if $dist(c_1, c_2) < s$ **then**
$o.c_2 = c_2$
On enlève c_2 à T_2
Break {On sort de la boucle FOR.}
end if
end for
end for

On crée un objet pour chaque cible du reliquat de t_2 :

for all $c \in T_2$ **do**

Création de o avec $o.c_2 = c$

On enlève c à T_2 {Pour la cohérence de la cardinalité des ensembles à la fin du traitement.}

end for

On calcule les paramètres des objets de O :

for all $o \in O$ **do**

$o.p = fus(o.c_1, o.c_2)$

end for

3.5 Le transport des informations

Les cartes de l'environnement doivent être disponibles à une fréquence suffisamment élevée pour permettre une bonne précision de la localisation. Comme nous pouvons le voir en 5.5, l'architecture asynchrone du module serveur permet de gérer au mieux les particularités temporelles des traitements. Cette partie s'intéresse à quantifier la volumétrie des messages transmis entre les modules de l'application et les bandes passantes nécessaires pour ces liaisons.

Dans un premier temps, nous présenterons les structures de données utilisées. En nous servant ensuite des caractéristiques de l'infrastructure, nous déterminerons les formules de calcul des bandes passantes et donnerons un exemple pour chacune des liaisons.

3.5.1 Les liaisons utilisées

La partie 5.5 nous présente le détail de l' architecture concernée par le temps réel. Dans cette version de *ParkView*, l'ensemble des traitements est supporté par la même machine en utilisant des segments de mémoire partagée pour l'échange d'information. Nous sommes donc dans le cas idéal. Les transmissions entre ces modules ne peuvent être plus rapides. Nous nous intéressons plutôt au cas où notre architecture sera décentralisée. A ce moment, les *trackers* seront supportés par des stations d'acquisition indépendantes.

Dans le chapitre 2, nous avons vu que nous utilisons un réseau dédié à notre application. Il supporte le trafic entre les stations d'acquisition et le serveur, et entre le serveur et le routeur hertzien destiné aux communications avec les clients *ParkView*.

Le tableau 3.1 liste ces liens et leurs caractéristiques.

Liaison	Vitesse	Nombre
Trackers - Serveur	$100Mb.s^{-1}$	n_t
Routeur hertzien - Clients *ParkView*	$11Mb.s^{-1}$	n_c
Serveur - Routeur	$100Mb.s^{-1}$	1

TAB. 3.1 – Caractéristiques des liens *ParkView*

3.5.2 Structure des données

Le tableau 3.2 donne la structure des données renvoyées par le *tracker* pour une cible. Elles sont mises à la disposition du module de fusion pour chaque rafraîchissement de l'image.

Libellé	Définition	Unité	Type
Id	Identificateur de la cible	-	4B
t_s	Date de la mesure	$seconde$	4B
$t_{\mu s}$	Date de la mesure	$\mu seconde$	4B
x	Abscisse de la cible dans l'image vidéo	$pixel$	2B
y	Ordonnée de la cible dans l'image vidéo	$pixel$	2B
$\dot{x}$	Composante de la vitesse	$pixel.s^{-1}$	2B
$\dot{y}$	Composante de la vitesse	$pixel.s^{-1}$	2B
xx	Param. ellipse englobante - x	-	2B
yy	Param. ellipse englobante - y	-	2B
xy	Param. ellipse englob. - angle	-	2B
Total			26B

TAB. 3.2 – Structure des données renvoyées par le *tracker*

Le tableau 3.3 présente la structure des données renvoyées par le serveur pour un objet de l'environnement.

3.5.3 Calcul des débits

Liaison trackers↔serveur

Les futures liaisons entre les *trackers* et le serveur se feront sur une branche de réseau spécialisée. Il faut toutefois garder à l'esprit que même si la plupart des réseaux sont maintenant commutés, l'ensemble des *trackers* s'adressera au même serveur. Il y aura donc bien un multiplexage temporel au niveau du lien commutateur ↔serveur. La bande passante sera donc partagée entre les *trackers*.

La formule générale pour calculer la bande passante utilisée par les *trackers* est :

Libellé	Définition	Unité	Domaine de valeur	Type
Id	Identificateur de l'objet	-	[0 ; 4294967296]	4B
t_s	Date de la mesure	s	[0 ; 4294967296]	4B
$t_{\mu s}$	Date de la mesure	μs	[0 ; 999 999]	4B
x	Abscisse de la cible dans le plan de référence	mm	[0 ; 16777215]	3B
y	Ordonnée de la cible dans le plan de référence	mm	[0 ; 16777215]	3B
$\dot{x}$	Composante de la vitesse	$cm.s^{-1}$	[-32768 ; 32767]	2B
$\dot{y}$	Composante de la vitesse	$cm.s^{-1}$	[-32768 ; 32767]	2B
xx	Param. ellipse englobante - x	-	-	2B
yy	Param. ellipse englobante - y	-	-	2B
xy	Param. ellipse englob. - angle	-	-	2B
θ	Angle (non utilisé)	$\frac{1}{1000}rad$	[0 ; 6283]	2B
$\dot{\theta}$	Vitesse angulaire (non utilisé)	$\frac{1}{1000}rad.s^{-1}$	[-32768 ; 32767]	2B
Total				32B

TAB. 3.3 – Structure des données renvoyées par le serveur

$$b = f.\sum_{i=0}^{n_t}(s_h + n_{c_i}.s_c)$$

avec :

- b, la bande passante en $b.s^{-1}$,
- s_h, la taille de l'ensemble des entêtes de protocole en bit,
- s_c la taille des données pour une cible en bit,
- f, la fréquence de rafraîchissement des données en Hertz,
- n_t, le nombre de *trackers*,
- n_{c_i}, le nombre de cible pour le i^{eme} *tracker*.

A titre d'exemple, en prenant :

- $s_h = 428$, si nous nous reportons à l' annexe B et à [Com00], en faisant la somme des tailles des entêtes des protocoles *Ethernet / IP / TCP*. Ce volume est incompressible et nous devons en tenir compte quelque soit la nature des données transmises.
- $s_c = 26 \times 8 = 208$, taille des données transmises par un *tracker* pour une cible.
- $f = 25$, fréquence de rafraîchissement des images vidéo du système *PAL*.
- $n_t = 10$, le nombre de *trackers* nécessaires pour couvrir l'ensemble du parking de l'*INRIA Rhône-Alpes*.
- $\forall i, 1 \leq i \leq n_t, n_{c_i} = 20$, un nombre de cibles important pour chaque *tracker,*

le débit nécessaire est de :

$$b = 25 \times 10 \times (428 + 20 \times 208) = 1\,147\,700\ b.s^{-1} = 1120.12\ Kb.s^{-1}.$$

Le réseau utilisé a une bande passante maximum de 100 $Mb.s^{-1}$. En cas de charge, même en ne disposant que de $\frac{1}{10}^{eme}$de sa capacité, soit 10240 $Kb.s^{-1}$, nous pouvons encore faire transiter les informations issues des *trackers*.

Liaison clients $\leftrightarrow$serveur

Dans cette partie, nous ne nous intéressons qu'à la bande passante utilisée par les clients *ParkView* déjà connectés au serveur. Celle utilisée par les trames de connexion, c'est-à-dire, lors de l'abonnement au service d'un nouveau client, est ponctuelle et donc négligeable.

Il y a deux façons de transmettre les cartes aux clients. La première est de le faire nominativement, c'est-à-dire qu'une trame de données contenant une carte sera adressée à un client particulier. La deuxième, la plus judicieuse, utilise le mode *broadcast* des protocoles *Ethernet* et *IP*. Sachant qu'à un même moment, une carte de l'environnement est identique quelque soit le client, ce mode de transmission permet de n'envoyer qu'une trame vers l'ensemble des clients.

La formule générale pour calculer la bande passante utilisée entre le serveur de cartes et l'ensemble des clients est :

$$b = f.(s_h + n_o.s_o)$$

avec :

- n_o, le nombre d'objets dans l'environnement,
- s_o, la taille des données pour un objet.

A titre d'exemple, en prenant $n_o = 20$ et $f = 10$, soit un rafraîchissement de la carte toutes les 100 millisecondes, nous obtenons :

$$b = 10 \times (428 + 20 \times 256) = 55480\ b.s^{-1} = 54.18\ Kb.s^{-1}.$$

Le système de transmission utilisé est un réseau *Ethernet* hertzien. Sa bande passante est de 11 $Mb.s^{-1}$. Dans le pire des cas, alors que ce réseau nous est dédié, si nous ne disposions que de $\frac{1}{10}^{eme}$de celle-ci, cela représenterait encore 1126 $Kb.s^{-1}$. Soit une bande passante nous permettant d'augmenter très largement la fréquence de rafraîchissement de nos cartes et le nombre d'objets suivis.

Chapitre 4

Infrastructure matérielle

Une part importante de la conception de l'application *ParkView* concerne le développement de son infrastructure matérielle. Elle comprend la caractérisation des besoins, l'étude des spécifications des équipements souhaités, la sélection de ceux-ci et de leurs fournisseurs. L'ensemble de ces tâches dépasse le cadre de la conception logicielle mais fait pourtant partie du quotidien de l'ingénieur en informatique. Il doit en effet, au-delà de sa propre technologie, être sans cesse capable d'appréhender de nouveaux domaines de compétence afin d'être le chef d'orchestre de projets plus globaux.

Dans le cas de *ParkView*, les connaissances demandées concernent l'optique, la robotique, l'électricité, le dessin industriel et la coordination de différents corps de métier.

4.1 Choix des capteurs

L'application *ParkView* utilise le système de suivi de cible développé par l'équipe *Prima*. Celui-ci est capable de travailler à partir de n'importe quelle source vidéo. Il utilise pour cela des cartes d'acquisition compatibles avec les *drivers* `v4l` pour le système Linux. Dans le cadre de *ParkView*, nous utiliserons donc des caméras vidéo. Le marché offre le choix entre un grand nombre de modèles. Ceux-ci vont de la caméra *N&B* basse résolution jusqu'aux caméras numériques couleurs de laboratoire. Les travaux menés sur la vision sont souvent basés sur l'étude de la couleur dans l'image, nous porterons donc notre choix sur une caméra couleur. En outre, les modèles sélectionnés devront pouvoir être synchronisables.

4.1.1 Caméras numériques

Depuis quelques temps, des caméras à transmission numérique, respectant la norme *IEEE*-1394, sont disponibles sur le marché. Ces modèles offrent une définition d'image importante et sont compatibles avec la plupart des systèmes dont *Linux*. Néanmoins, la norme *IEEE*-1394

ne permet pas d'utiliser des liaisons de plus de 4.5 mètres sans utiliser de répéteurs. Comme nous le verrons, l'application *ParkView* utilise plusieurs liaisons vidéo de 20 et 35 mètres. Pour des raisons de coût et de fiabilité, il n'est donc pas possible d'utiliser ce type de transmission. Nous avons donc privilégié les caméras à transmission analogique.

4.1.2 Caméras analogiques

Plusieurs contraintes sont à prendre en compte pour le choix de la caméra. Le modèle choisi doit être utilisable en extérieur. Il doit donner la possibilité de contrôler ses paramètres internes à distance. Ceci permet, par exemple, de fixer l'ouverture de l'iris afin que les systèmes de suivi de cibles ne soient pas perturbés par des changements d'intensité lumineuse automatiques intempestifs.

Le modèle choisi [JVC01] est destiné à la vidéo-surveillance, il est conçu pour fonctionner en extérieur en utilisant un boîtier de protection thermostaté. Il est équipé d'une interface série à la norme *RS*-485 permettant de contrôler ses paramètres internes à plusieurs centaines de mètres de distance en utilisant le protocole décrit dans [JVC02]. Enfin, il est équipé d'une entrée de synchronisation externe permettant de contrôler le moment d'acquisition des images.

4.1.3 Caractérisation des objectifs utilisés

Des caractéristiques des objectifs va dépendre la couverture de parking prise en compte par l'application *ParkView*. Il nous faut tout d'abord déterminer ces zones. La première version de *ParkView* est située sur le parking arrière de l'*INRIA*. En reprenant les plans de conception du bâtiment et en effectuant des mesures sur le terrain, nous connaissons l'ensemble de ses dimensions. La figure 4.1, présente un schéma de côté du parking équipé de *ParkView*.

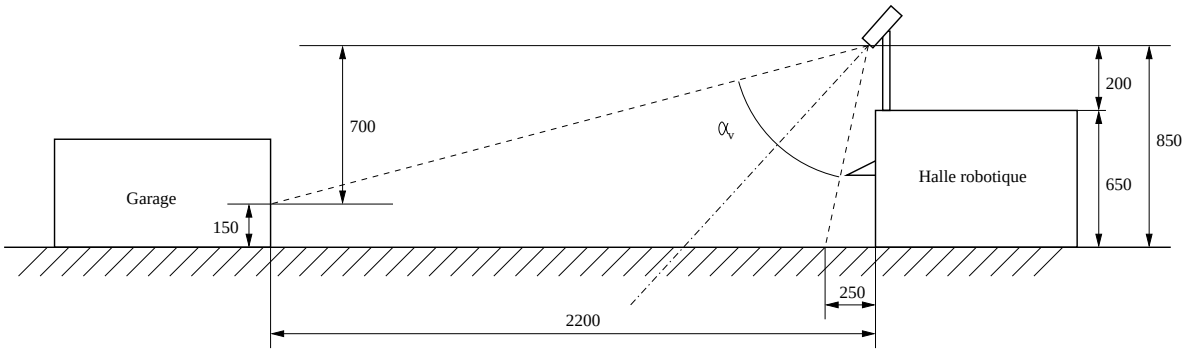


FIG. 4.1 – Schéma du parking, vue de côté

Nous avons fixé les caractéristiques suivantes pour l'infrastructure de *ParkView* sur le parking :

- la hauteur des poteaux supportant les caméras est fixée à deux mètres. Bien que nous sachions que les caméras doivent être placées le plus haut possible pour limiter les effets de la perspective, il nous a fallu trouver un compromis entre un résultat idéal et la praticité des manipulations. Etant dans un contexte de recherche, il est fréquent de devoir modifier le positionnement des caméras. Leur accès doit être facile et ne pas poser de problèmes de sécurité.
- le champ de vision vertical de la caméra arrive à une hauteur de 1,5m au niveau du garage. Ceci permet la détection des personnes ou des véhicules évoluant le long de celui-ci,
- nous excluons une bande de 2,5 mètres de large, le long de la halle robotique, qui est cachée par un auvent.

A partir de ces données, nous calculons l'angle de vision vertical minium α_v des caméras installées sur la halle robotique :

$$\alpha_v = tan^{-1}(\frac{2200}{700}) - tan^{-1}(\frac{250}{850}) \approx 55,96°.$$

A partir de α_v et en ayant connaissance de la dimension des capteurs de nos caméras, nous pouvons déterminer la longueur focale des objectifs à utiliser.

Géométrie de la caméra

La figure 4.2 représente le schéma caractéristique de la géométrie d'une caméra.

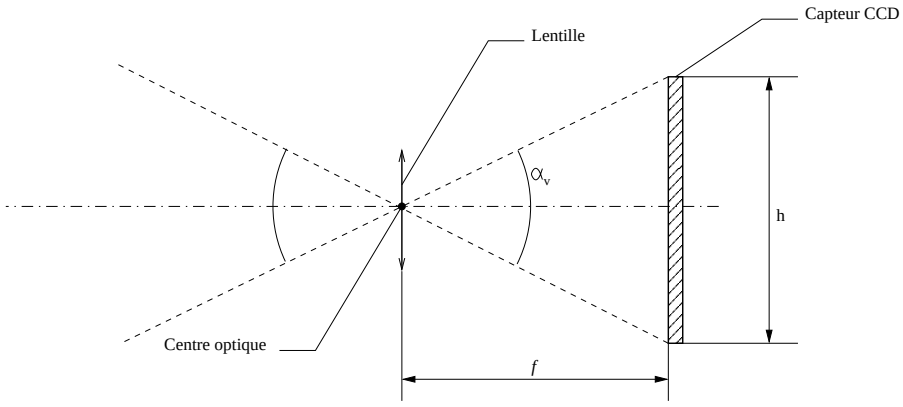


FIG. 4.2 – Géométrie de la caméra

La longueur focale f est la distance entre le plan de projection, ici le capteur CCD, et le centre optique de l'objectif. Les rayons lumineux passant par le centre optique ne sont pas

déviés par les lentilles de l'objectif[1].

L'équation (4.1) nous permet de calculer la longueur focale à utiliser à partir de la hauteur h de notre capteur CCD :

$$f = \frac{h}{2.\tan\frac{\alpha_v}{2}}. \tag{4.1}$$

Le tableau 4.1 nous renseigne sur les dimensions des capteurs les plus courants.

Capteur	l	h	d
1/3'	4.8 mm	3.6 mm	6.0 mm
1/2'	6.4 mm	4.8 mm	8.0 mm
2/3'	9.6 mm	7.2 mm	12.0 mm

TAB. 4.1 – Dimensions des capteurs courants

On remarque que la longueur des diagonales données ne correspond pas à la dénomination des capteurs donnée en pouce. Il faut savoir que cette dénomination est une valeur théorique et que la surface utile est toujours plus petite.

A partir des spécifications de la caméra [JVC01] et en appliquant l'équation (4.1), nous pouvons déterminer la longueur focale à utiliser :

$$f = \frac{4.8}{2.\tan\frac{55.96}{2}} = 4.51mm.$$

Il faut savoir que les objectifs offrant un tel angle de vue souffrent, en général, d'une importante distorsion optique. Celle-ci sera corrigée par le logiciel, voir la partie 3.3.2.

4.1.4 Validation des caractéristiques calculées

Afin de valider les caractéristiques des objectifs, nous avons réalisé une modélisation en trois dimensions du parking à l'aide du logiciel *AC3D* [Ini03]. La figure 4.3 nous présente une vue générale du parking équipé de deux caméras sur leur support.

Le logiciel de modélisation nous permet de placer la caméra virtuelle en lieu et place des caméras réelles et de faire varier son angle de vision horizontal α_h.

En sachant que le rapport $\frac{largeur}{hauteur}$ des capteurs est de $\frac{4}{3}$, le passage de α_v à α_h se fait par l'équation :

$$\alpha_h = \tan^{-1}\frac{4\times\tan\alpha_v}{3}.$$

[1] En théorie. Dans les faits, la distorsion optique dévie les rayons passant par le centre optique. Nous n'en tenons pas compte ici car les constructeurs l'intègrent dans le calcul de la longueur focale de leurs objectifs.

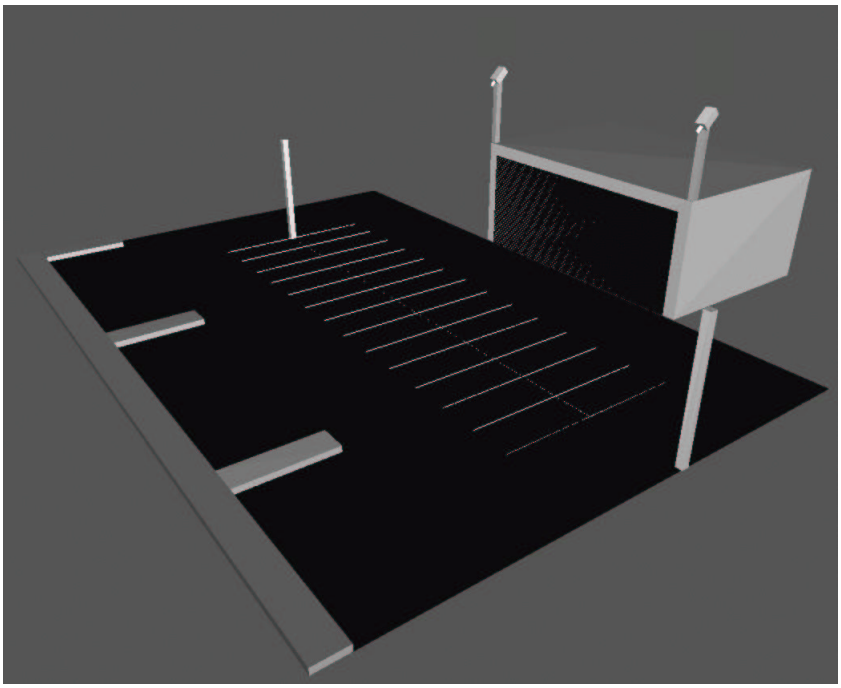


FIG. 4.3 – Modélisation 3D du parking équipé de ParkView

dans notre cas :

$$\alpha_h = \tan^{-1} \frac{4 \times \tan 55.96}{3} = 63°.$$

La figure 4.4 montre le résultat de la simulation pour un angle $\alpha_h = 63°$.

Cette simulation donne à l'image le résultat souhaité.

4.2 Banc de test

Avant que l'installation de *ParkView* sur le parking de l'*INRIA* soit effective, nous avons mis en place un banc de test, visible sur la figure 4.5. Celui-ci a été réalisé à l'échelle, c'est-à-dire, que le positionnement des caméras ainsi leur orientation respecte les paramètres précédemment définis.

La figure 4.6 représente les champs de vision des caméras projetés sur le plan du banc de test. Le grille de calibration est placée dans la zone de vision commune pour les tests de fusion de données.

Les deux images de la figure 4.7 donnent un aperçu des points de vue des caméras.

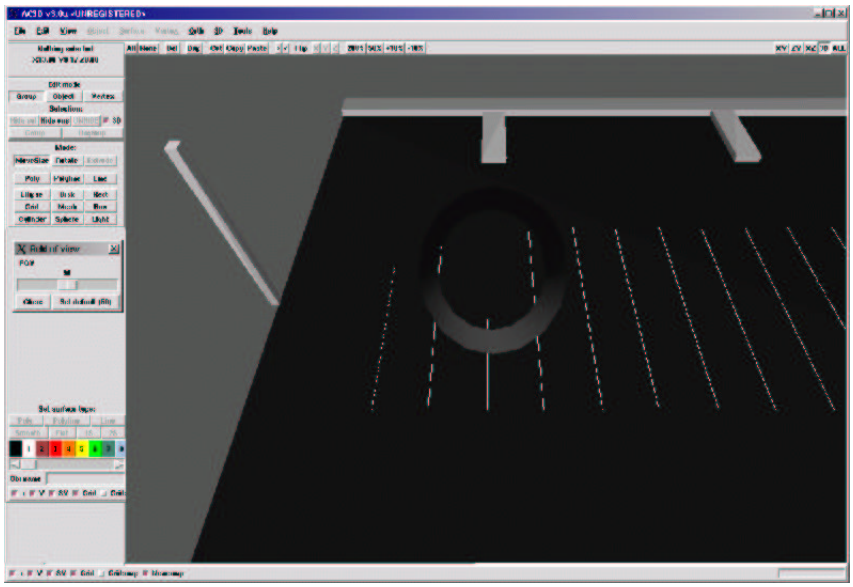


FIG. 4.4 – Simulation 3D de la vue du parking à partir d'une caméra pour $\alpha_h = 63°$

FIG. 4.5 – Banc d'essai ParkView

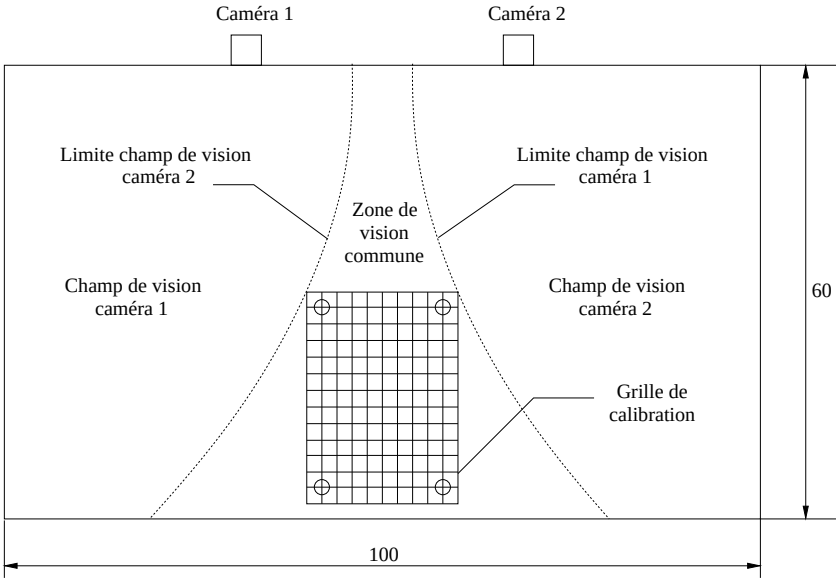
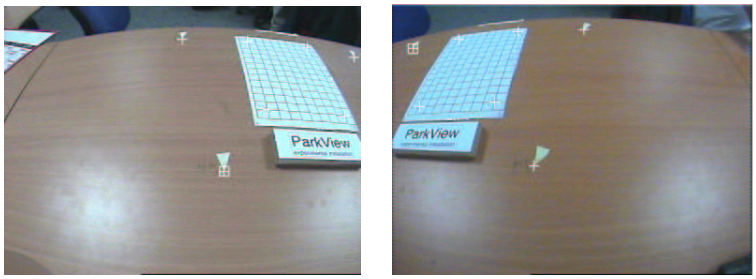


FIG. 4.6 – Champs de vision des caméras sur le banc de test



Caméra 2 Caméra 1

FIG. 4.7 – Points de vue depuis les caméras du banc de test

4.3 Equipement du bâtiment

4.3.1 Conception des poteaux

L'infrastructure *ParkView* utilise des poteaux supportant les caméras. Plus les caméras sont placées en hauteur, mois les effets de la perspective influencent le suivi des objets. Néanmoins, nous avons vu que les caméras doivent être accessibles facilement. Nous avons donc opté pour une hauteur de deux mètres afin d'y accéder sans équipements particuliers.

La figure 4.8 présente le plan ayant servi à la fabrication des poteaux. Ceux-ci sont équipés d'une embase carrée de fixation au toit de la halle robotique. Ils disposent d'une platine horizontale à leur sommet destinée à recevoir des équipements. L'équipe *Movi* a prévu d'y installer une caméra orientable Pan/Tilt destinée aux recherches sur le suivi rapproché de cibles. Sur chaque poteau, une caméra est fixée à l'aide d'un collier, le long du cylindre principal.

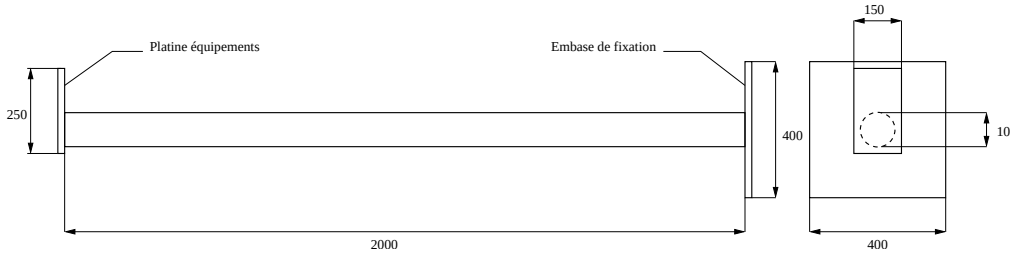


FIG. 4.8 – Schéma des poteaux supportant les caméras

4.3.2 Connectique

Câbles vidéos

Chaque poteau est équipé de cinq connexions vidéo. Deux sont utilisées pour le transport des images, deux autres sont utilisées pour la synchronisation. La dernière est une connexion de secours utilisée en cas de défaillance de l'une des quatre premières.

A partir des plans du bâtiment, les longueurs suivantes ont pu être déterminées :

- 5 câbles coaxiaux 75 Ohms de 35 mètres,
- 5 câbles coaxiaux 75 Ohms de 20 mètres.

Liaisons de commande

Chaque poteau est équipé de trois doubles paires torsadées pour les liaisons *RS*-485. Il y en a une par caméra ainsi qu'une en secours, soit :

- 3 câbles doubles paires torsadées de 35 mètres,

– 3 câbles doubles paires torsadées de 20 mètres.

Liaisons Ethernet

Les poteaux sont équipés pour être reliés au réseau de l'*INRIA* en vue de l'utilisation de futurs équipements. Il existe en effet, outre les nouvelles caméras à transmission numérique, des équipements disposant de serveur *TCP/IP* interne capable de mettre à disposition directement sur le réseau des informations. Dans le futur, nous pourrons ainsi piloter les caméras en s'adressant à leur serveur interne.

Afin de préparer notre infrastructure à ces équipements, nous avons doté chaque poteau d'une liaison réseau soit :

– 1 câble *Ethernet* de 35 mètres,
– 1 câble *Ethernet* de 20 mètres.

Alimentation

Chaque poteau est alimenté en 220V pour l'ensemble de ses équipements.

Passage des câbles

Le passage des câbles à travers le toit de la halle robotique est assuré par des cols de cygnes qui assurent l'étanchéité, voir figure 4.9. Leur diamètre permet d'étendre facilement *ParkView* par l'adjonction de nouveaux câbles. L'utilisation de deux cols de cygne permet de séparer les lignes de données (courant faible) des alimentations (courant fort) afin de limiter les perturbations électriques.

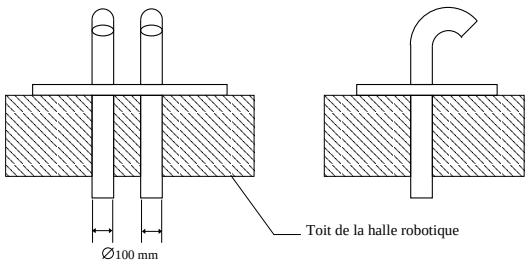


FIG. 4.9 – Cols de cygne

Chapitre 5

Développements logiciels

Ce chapitre est consacré à l'infrastructure logicielle de l'application *ParkView*. Celle-ci est divisée en deux parties : le paramétrage et l'exploitation.

L'ensemble des développements ont été réalisés en *C++* [Str97] en s'adjoignant des bibliothèques telles que *FLTK* [SES02] pour la gestion de l'interface graphique, *Gandalf* pour le traitement d'images [McL99], ainsi que [Fou01] pour la gestion des fichiers *XML*.

Ce chapitre utilise le formalisme *UML* pour ses schémas dont on trouvera une description dans [FL00, FS00].

Dans une première partie, nous présentons l'architecture globale du logiciel *ParkView*. Nous introduirons ensuite les concepts particuliers de son paramétrage en décrivant l'interface graphique réalisée puis le module `ParkviewApp` modélisant les structures de données. Après avoir décrit l'architecture et les particularités du paramétrage du *tracker Prima*, nous détaillerons le module `ParkviewMapServ` réalisant à la fois la fusion des données et le serveur de carte. Enfin, nous décrirons un test de fusion effectué.

5.1 Architecture globale

L'application *ParkView* a été voulue modulaire dans sa conception. La figure 5.1 permet de s'en rendre compte en présentant l'ensemble des dépendances entre les modules. Le schéma sépare les développements propres à *ParkView* des autres modules utilisés.

Plusieurs raisons ont motivé cette conception. Du point de vue du génie logiciel, il est beaucoup plus simple de concevoir et de mettre en place une architecture lorsque celle-ci est divisée en modules. Chacun d'eux prend alors en charge les traitements et les structures de données propres à une problématique. Cette problématique pouvant être commune à plusieurs applications, la réutilisabilité du code prend alors tout son sens. Il n'est possible d'assurer cette réutilisabilité que si le module est considéré comme autonome et qu'il a pu bénéficier de

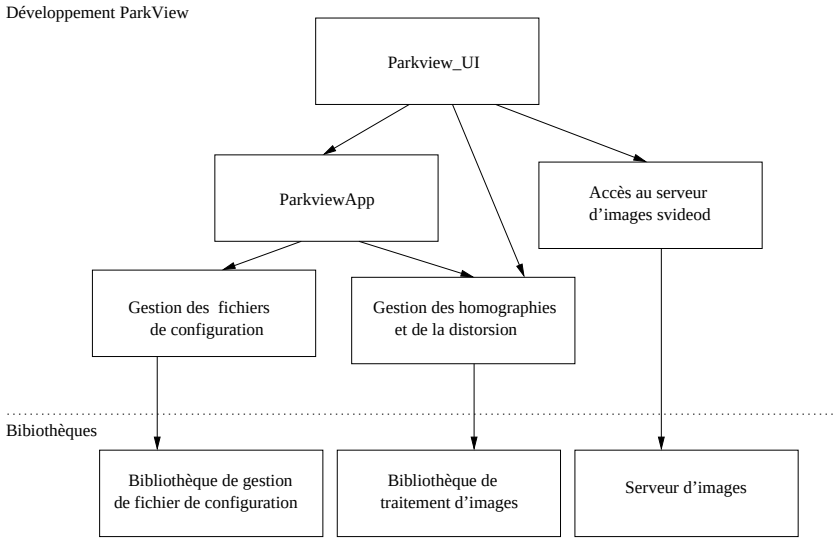


FIG. 5.1 – Schéma de dépendance des modules pour l'application de paramétrage

tests et de validations indépendants.

Dans le cas de *ParkView*, outre la volonté de réutilisabilité, c'est la dépendance vis-à-vis d'une technologie qui a motivé cette conception. Ainsi, par l'intermédiaire de la société d'exploitation *Blue Eye Video*, l'équipe *Prima* participe aux développements des produits de suivi de cibles vidéo. Certains modules de l'application *ParkView* concernant ces technologies pourront, à terme, être intégrés à ces produits. Certaines bibliothèques libres de droits peuvent difficilement être utilisées dans un contexte commercial. Il est donc nécessaire de séparer au mieux les différentes parties du logiciel afin de pouvoir les inter-changer ou les intégrer indépendamment dans de nouveaux environnements.

La figure 5.2 illustre la possibilité pour des applications de venir s'interfacer à différents niveaux de *ParkView*.

5.2 L'interface graphique `Parkview_UI`

L'interface de paramétrage de *ParkView* a pour but essentiel de gérer une base de données de points remarquables dans les plans appelés *landmark* ainsi que les instances de ces points dans les différents plans. Ces points de correspondance seront ensuite utilisés pour le calcul des transformations homographiques.

Le paramétrage s'effectue à l'aide d'une interface graphique qui se veut intuitive. Dans bien des cas, il est nécessaire de sélectionner des points particuliers dans l'image. A l'aide d'une interface graphique, il est plus facile de les pointer directement à l'aide d'une souris plutôt que

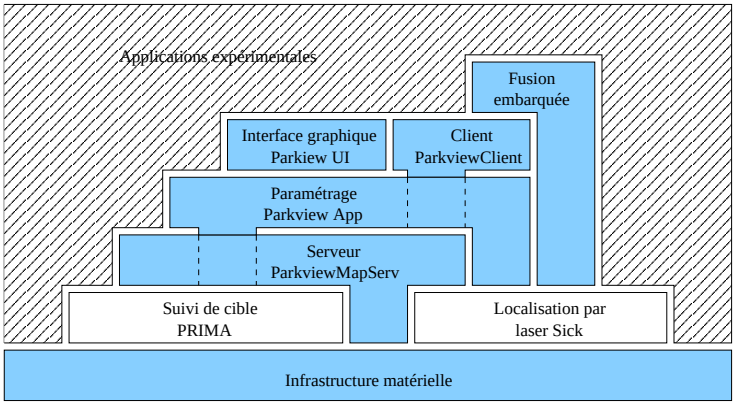


FIG. 5.2 – Interfaces des modules *ParkView*

de saisir leurs coordonnées.

Bien que dans sa version actuelle, l'application *ParkView* ne possède que deux caméras, son interface est un MDI[1] et est déjà donc prête pour un plus grand nombre. Il en est de même pour la gestion des fichiers de cartes.

Nous nous intéresserons dans cette partie à la description de l'interface graphique en déroulant le processus type de paramétrage. Dans un premier temps, nous décrirons l'interface principale gérant le plan de référence. Nous nous intéresserons ensuite à la prise en compte de nouveaux plans. Dans le cas des plans vidéo, nous verrons comment prendre en compte la distorsion optique. Enfin, nous calculerons les paramètres d'une homographie et nous exporterons cette configuration.

5.2.1 La fenêtre projet, plan de référence

La fenêtre de projet est la première qui apparaît dès le lancement de l'application, voir figure 5.3.

La première opération à effectuer est de créer un nouveau projet. Cela se fait en utilisant le menu *Projet* visible sur la figure 5.4.

Une fois un projet créé, les menus *Vidéo* et *Map* gérant les plans vidéo et les cartes sont accessibles.

La fenêtre principale permet de gérer les instances des *landmarks* dans le plan de référence. Pour en ajouter, on sélectionne *Add Landmark*, la fenêtre de création / association présentée en figure 5.5 apparaît. Les coordonnées des points dans le plan de référence sont saisis à l'aide du clavier. Les instances de *landmarks* apparaissent alors sous forme de croix. Lorsque l'on

[1] Une interface MDI (Multi Documents Interface) permet d'ouvrir plusieurs fenêtres du même type avec des données différentes dans la même application.

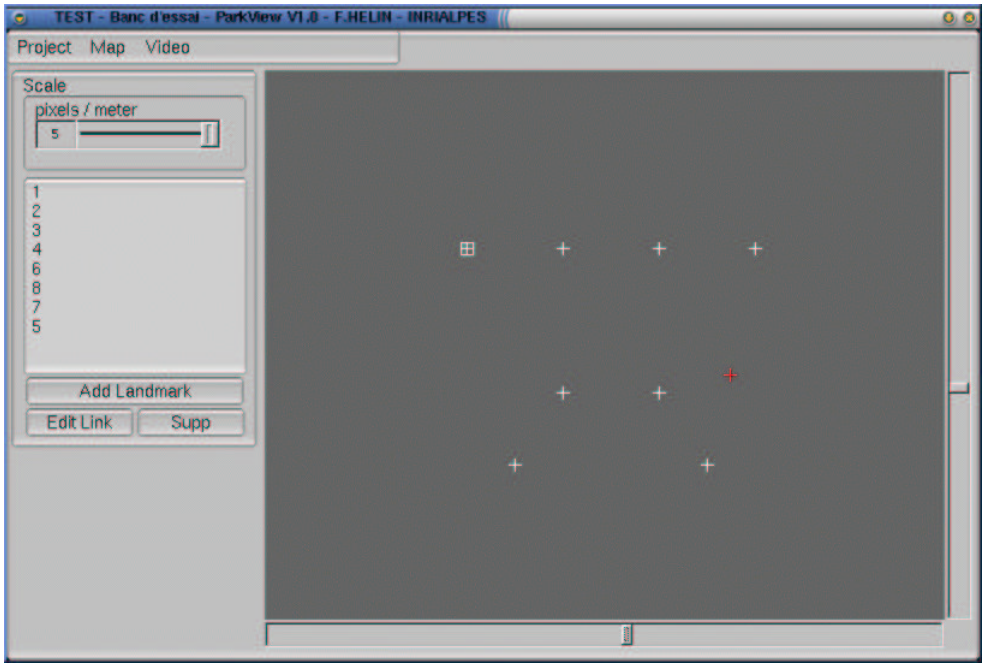


FIG. 5.3 – Fenêtre de projet, plan de référence

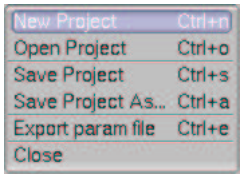


FIG. 5.4 – Menu de gestion de projet

ajoute de nouveaux libellés de *landmarks* dans la fenêtre de création / association, ceux-ci sont ensuite disponibles pour tous les plans. On peut noter qu'il est possible de faire varier l'échelle de représentation des points à l'aide du curseur horizontal *scale*.

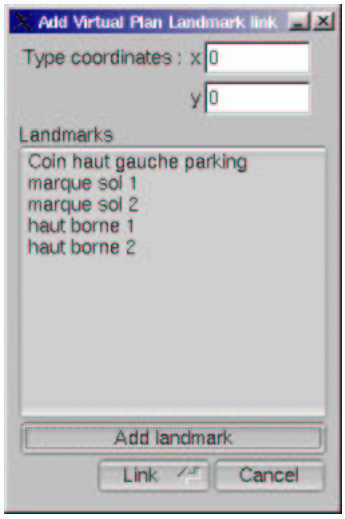


FIG. 5.5 – Fenêtre d'association / création de landmarks dans le plan de référence

Le menu *Map* permet d'accéder aux cartes du projet. Il permet également d'en ajouter, en leur attribuant un libellé, voir figure 5.6.

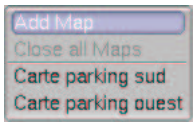


FIG. 5.6 – Menu de gestion des cartes, ajout d'une carte

5.2.2 Gestion des cartes

La figure 5.7 présente la fenêtre de gestion des cartes.

Actuellement, les cartes servent uniquement à vérifier que les instances de *landmark* sont saisies sans erreur. Pour cela, il est possible de parcourir la carte à l'aide de la souris et de visualiser ce parcours sur le plan de référence.

L'ajout des instances de *landmark* dans une carte se fait directement, en cliquant sur celle-ci. La fenêtre d'agrandissement présentée en figure 5.8 apparaît alors.

Cette fenêtre offre plus de précision dans le pointage des *landmarks* sur la carte. En cliquant dans la fenêtre de zoom, la fenêtre d'association instance / *Landmark* apparaît, voir figure 5.9.

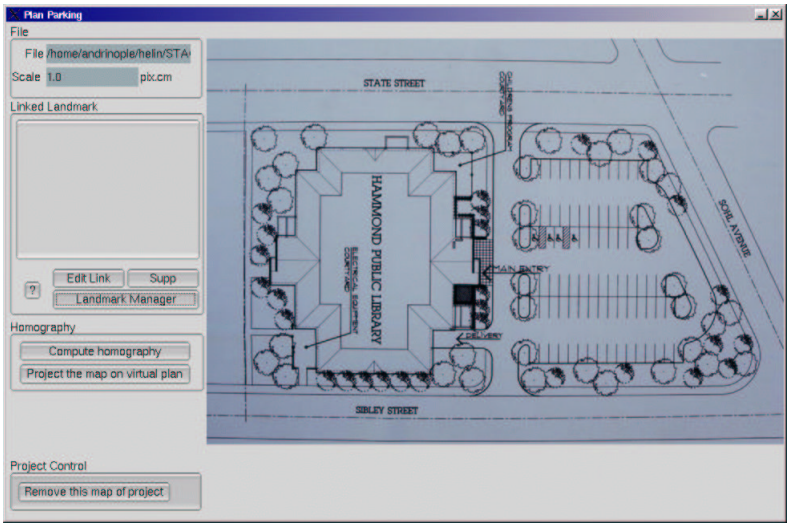


FIG. 5.7 – Fenêtre de gestion des cartes

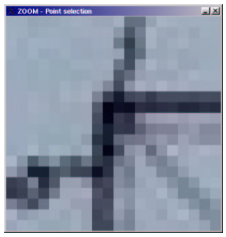


FIG. 5.8 – Fenêtre de zoom sur carte

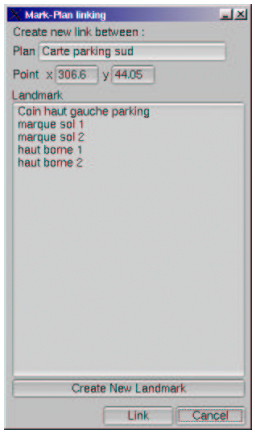


FIG. 5.9 – Fenêtre de gestion des *landmarks* et de leurs instances dans une carte

Les *landmarks* sont directement représentés sur la carte, voir figure 5.10.

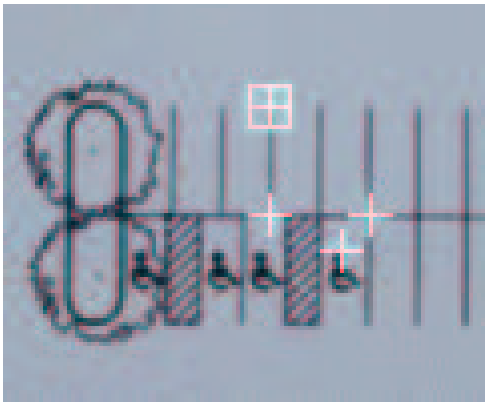


FIG. 5.10 – Exemple de représentation de *landmarks* dans une carte

5.2.3 Gestion des plans vidéo

L'ajout d'un nouveau plan vidéo au projet se fait à l'aide du menu *Video* dans la fenêtre de projet. L'interface demande de spécifier successivement le libellé du plan puis la référence du *buffer* vidéo *shmKey*. Cette référence est celle qui a été utilisée lors du lancement du serveur vidéo *Prima* et permet de sélectionner l'une des caméras reliées à la station de travail.

La fenêtre de gestion de plans vidéo, présentée en figure 5.11, est d'une utilisation comparable à celle des cartes. Il est ainsi possible de positionner des instances de *landmarks* dans l'image via la fenêtre de zoom.

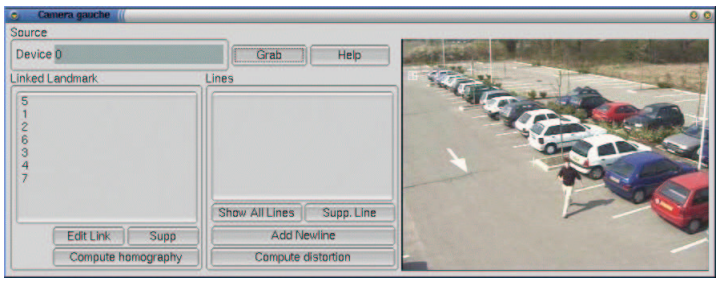


FIG. 5.11 – Fenêtre de gestion de plans vidéo

Correction de la distorsion

La correction de la distorsion s'effectue en désignant des points dans l'image comme devant être alignés. La figure 5.12 présente les points désignés reliés par des segments ainsi que les

références des droites. Le test présenté a été effectué avec des feuilles placées devant l'une des caméra du banc de test. On remarque que les effets de la distorsion sont particulièrement visibles à la périphérie de l'image.

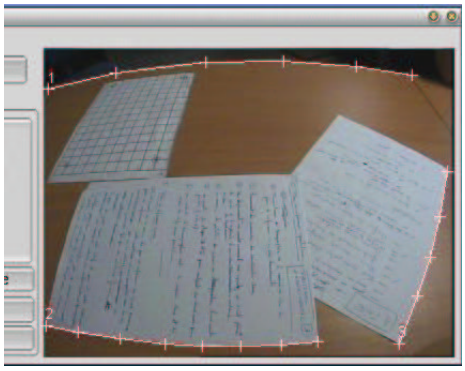


FIG. 5.12 – Marquage des lignes pour la correction de la distorsion

Une fois que les droites sont tracées, la correction s'effectue en cliquant sur le bouton "compute disto.". Cette correction n'est pas obligatoire pour le calcul de l'homographie. En effet, lorsqu'on utilise des caméras à longue focale, les effets de la distorsion sont négligeables, il n'est donc pas nécessaire de la corriger. Dans la figure 5.11, la prise de vue a été effectuée avec une caméra équipée d'un objectif de longueur focale plus importante. On remarque que la rectitude du trottoir, dans la partie supérieure droite de l'image, n'apparaît pas modifiée. Ici, la correction n'est pas nécessaire.

5.2.4 Calcul des homographies

Dans chaque fenêtre de gestion des plans, il y a un bouton "compute homographie". Il ne peut être utilisé que s'il existe au moins quatre instances de *landmark* communes dans le plan sélectionné et celui de référence. Si ce n'est pas le cas, l'interface graphique le signale par un message d'erreur. Si les instanciations sont correctement faites, les coefficients de la matrice de transformation homographique sont pris en compte par le logiciel.

Dans le cas des plans vidéo, l'opération tient compte de l'éventuelle correction de distorsion en amont, en utilisant les coordonnées corrigées des instances de *landmark* pour le calcul des homographies.

5.2.5 Génération du fichier de paramètres

Le fichier de paramètres à destination des *trackers* et du serveur de cartes est généré à partir de l'option "export param file" du menu de gestion de projet présenté en figure 5.4.

Exemple de fichier *XML* de paramétrage :

```
# --- TEST 4 - TEST BED/SUIVI BALLE TENNIS ---
<parkview>
  <video id = 2>
    <label  value = "Camera gauche"/>
    <shmkey value = "11111"/>
    <homography>
      <m11 value =-0.227695/>
      <m12 value =0.127981/>
      <m13 value =67.717/>
      <m21 value =-0.0428032/>
      <m22 value =0.387409/>
      <m23 value =-4.66079/>
      <m31 value =-0.0014648/>
      <m32 value =0.00480395/>
      <m33 value =1/>
    </homography>
  </video>
  <video id = 3>
    <label  value = "Camera droite"/>
    <shmkey value = "22222"/>
    <homography>
      <m11 value =-0.470616/>
      <m12 value =-0.143822/>
      <m13 value =67.3288/>
      <m21 value =0.103593/>
      <m22 value =0.890681/>
      <m23 value =-42.0701/>
      <m31 value =0.00358595/>
      <m32 value =0.011528/>
      <m33 value =1/>
    </homography>
    <distortion>
      <cx value = 189.5424/>
      <cy value = 245,81789/>
      <k1 value = 0.1354354/>
      <k2 value = 1.4547484/>
    </distortion>
  </video>
</parkview>
```

On remarque dans le fichier que les coefficients des homographies sont associés aux deux plans vidéo. On remarque également les paramètres de la distorsion qui n'ont été calculés que pour le plan 3.

5.3 Le module de paramétrage `ParkviewApp`

Le module `ParkviewApp` prend en charge l'ensemble des structures de données propre au paramétrage de l'application. Ceci concerne les plans utilisés (cartes ou images vidéo), la distorsion optique des caméras, les calculs homographiques, l'exportation des données de paramétrage vers le module d'exploitation ainsi que la sauvegarde des projets. La figure 5.13 présente son diagramme *UML* de classe.

5.3.1 La classe *P_Project*

La classe *P_Project* est centrale au module `ParkviewApp`. Elle permet de gérer les collections de données de l'application. Lorsqu'un objet est instancié par l'interface graphique, sa référence doit être portée à la connaissance de l'unique objet de type `P_Projet` à l'aide des classes `add.....` . L'algorithme suivant décrit la procédure classique de création et de prise en compte d'un nouvel objet dans `ParkviewApp` :

```
AfficheFenetreCreationObjet(param);
MaClasse nouvelObjet = new MaClasse(param);
clefNouvelObjet = monprojet.addObjet(nouvelObjet);
```

Construction des objets *P_Projet*

Méthode	Description	Retour d'erreur
`P_Project (const char *name_)`	Constructeur	-
`~P_Project ()`	Destructeur	-

Méthodes de gestion de la sauvegarde du projet

Un projet est définit par son libellé et par le nom de son fichier de sauvegarde. Ce fichier est au format *XML* et utilise la bibliothèque *Xerces* [Fou01].

Méthode	Description	Retour d'erreur

`int setfilename ()`	Définit le nom du fichier d'enregistrement du projet	-1 - Le nom est pris en compte, 0 - Ce fichier existe déjà, le nom est pris en compte.
`const char* getfilename ()`	Retourne le nom de fichier courant du projet ou NULL	-
`const char* getName ()`	Retourne le nom du projet	-

Méthodes privées de modification de projet

Ces méthodes sont utilisées dans l'implémentation des méthodes de `P_Project` et peuvent l'être dans celles des classes dérivées de `P_Project`.

Méthode	Description	Retour d'erreur
`void setIsModified ()`	Place l'indicateur interne de projet modifié à "vrai".	-
`int getIsModified ()`	Retourne l'état de modification du projet	-1 - Le projet a été modifié, 0 - Le projet n'a pas changé.

Méthodes d'accès séquentiel aux références d'objets

Le module `ParkviewApp` permet un accès séquentiel aux objets. Ceci permet de parcourir l'ensemble des objets sans avoir connaissance des mécanismes gérant les collections. L'implémentation de ce système de séquence est basée sur l'utilisation des conteneurs de la bibliothèque standard du *C++* [Str97, Chapitre III].

En *C++*, chaque objet possède une référence, son adresse, attribuée par le système. Lors des opérations de sauvegarde, cette adresse ne peut pas être prise en compte puisqu'elle dépend de la machine elle-même. Il faut donc gérer un système de clef indépendant de la plate-forme utilisée qui puisse maintenir la cohérence des liens entre objets.

Méthode	Description	Retour d'erreur

`T_ID_Mark initFetchMark ()`	Initialise la séquence	0 - Pas d'objet de la collection
`T_ID_Plan initFetchVideo ()`	Initialise la séquence	0 - Pas d'objet de la collection
`T_ID_MarkPlan initFetchMarkPlan ()`	Initialise la séquence	0 - Pas d'objet de la collection
`T_ID_Plan initFetchMap ()`	Initialise la séquence	0 - Pas d'objet de la collection
`P_Map* getCurrentMap ()`	Retourne un pointeur sur l'objet de type `P_Map` courant de la séquence	*NULL* - Pas d'objet courant.
`P_Mark* getCurrentMark ()`	Retourne un pointeur sur l'objet de type `P_Mark` courant de la séquence	*NULL* - Pas d'objet courant.
`P_Video* getCurrentVideo ()`	Retourne un pointeur sur l'objet de type `P_Video` courant de la séquence	*NULL* - Pas d'objet courant.
`P_MarkPlan* getCurrentMarkPlan ()`	Retourne un pointeur sur l'objet de type `P_MarkPlan` courant de la séquence	*NULL* - Pas d'objet courant.
`T_ID_Plan getCurrentMapId ()`	Retourne la clef de l'objet de type `P_Map` courant de la séquence	*NULL* - Pas d'objet courant.
`T_ID_Mark getCurrentMarkId ()`	Retourne la clef de l'objet de type `P_Mark` courant de la séquence	0 - Pas d'objet courant.
`T_ID_Plan getCurrentVideoId ()`	Retourne la clef de l'objet de type `P_Video` courant de la séquence	0 - Pas d'objet courant.
`T_ID_MarkPlan getCurrentMarkPlanId ()`	Retourne la clef de l'objet de type `P_Markplan` courant de la séquence	0 - Pas d'objet courant.
`P_Map* fetchMap ()`	Déplace le curseur sur l'objet de type `P_Map` suivant et retourne son adresse.	*NULL* - Pas d'objet suivant.
`P_Mark* fetchMark ()`	Déplace le curseur sur l'objet de type `P_Mark` suivant et retourne son adresse.	*NULL* - Pas d'objet suivant.
`P_Video* fetchVideo ()`	Déplace le curseur sur l'objet de type `P_Video` suivant et retourne son adresse.	*NULL* - Pas d'objet suivant.
`P_MarkPlan* fetchMarkPlan ()`	Déplace le curseur sur l'objet de type `P_MarkPlan` suivant et retourne son adresse.	*NULL* - Pas d'objet suivant.
`T_ID_Plan fetchMapId ()`	Déplace le curseur sur l'objet de type `P_Map` suivant et retourne sa clef.	0 - Pas d'objet suivant.
`T_ID_Mark fetchMarkId ()`	Déplace le curseur sur l'objet de type *P_Mark* suivant et retourne sa clef.	0 - Pas d'objet suivant.
`T_ID_Plan fetchVideoId ()`	Déplace le curseur sur l'objet de type `P_Video` suivant et retourne sa clef.	0 - Pas d'objet suivant.
`T_ID_MarkPlan fetchMarkPlanId ()`	Déplace le curseur sur l'objet de type `P_MarkPlan` suivant et retourne sa clef.	0 - Pas d'objet suivant.

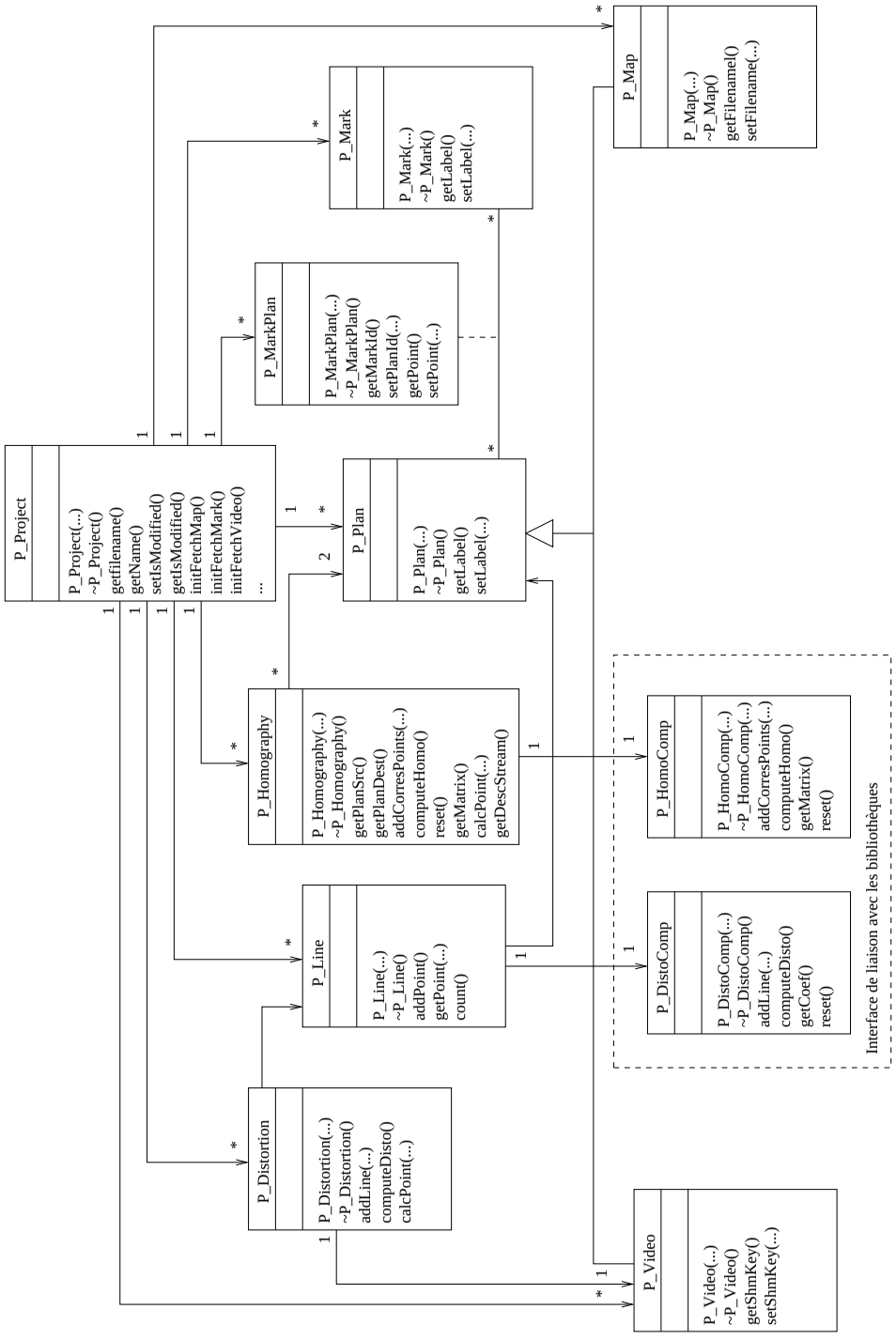


FIG. 5.13 – Diagramme *UML* des dépendances du module *ParkviewApp*

Méthode d'accès direct aux objets

Méthode	Description	Retour d'erreur
`P_Map* getMap (T_ID_Plan id_)`	Retourne un pointeur sur l'objet de type `P_Map`.	*NULL* - Pas d'objet ayant cette référence.
`T_ID_Plan getMap (P_Map* map_)`	Retourne la clef d'un objet `P_Map` à partir de son adresse.	*NULL* - Pas d'objet ayant cette référence.
`P_Mark* getMark (T_ID_Mark id_)`	Retourne un pointeur sur l'objet de type `P_Mark`.	*NULL* - Pas d'objet ayant cette référence.
`T_ID_Mark getMark (P_Mark* mark_)`	Retourne la clef d'un objet *P_Mark* à partir de son adresse.	0 - Pas de référence pour cet objet.
`P_Video* getVideo (T_ID_Plan id_)`	Retourne un pointeur sur l'objet de type `P_Video`.	*NULL* - Pas d'objet ayant cette référence.
`T_ID_Plan getVideo (P_Video* video_)`	Retourne la clef d'un objet `P_Video` à partir de son adresse.	0 - Pas de référence pour cet objet.
`P_Plan* getPlan (T_ID_Plan id_)`	Retourne un pointeur sur l'objet de type `P_Plan`.	*NULL* - Pas d'objet ayant cette référence.
`T_ID_Plan getPlan (P_Plan* plan_)`	Retourne la clef d'un objet `P_Plan` à partir de son adresse.	0 - Pas de référence pour cet objet.
`P_Homography* getHomo (T_ID_Homo id_)`	Retourne un pointeur sur l'objet de type `P_Homography`.	*NULL* - Pas d'objet ayant cette référence.
`P_Distortion* getDiso (T_ID_Disto id_)`	Retourne un pointeur sur l'objet de type `P_Distortion`.	*NULL* - Pas d'objet ayant cette référence.
`P_MarkPlan* getMarkPlan (T_ID_MarkPln id_)`	Retourne un pointeur sur l'objet de type `P_MarkPlan`.	*NULL* - Pas d'objet ayant cette référence.
`T_ID_MarkPlan getMarkPlan (P_MarkPlan* markplan_)`	Retourne la clef d'un objet `P_MarkPlan` à partir de son adresse.	0 - Pas de référence pour cet objet.

Méthodes de gestion des objets

Méthode	Description	Retour d'erreur
`T_ID_Plan addMap (P_Map* map_)`	Ajoute un objet de type `P_Map` au projet et retourne sa clef.	0 - L'ajout ne s'est pas effectué.
`T_ID_Mark addMark (P_Mark* mark_)`	Ajoute un objet de type `P_Mark` au projet et retourne sa clef	0 - L'ajout ne s'est pas effectué.
`T_ID_Plan addVideo (P_Video* video_)`	Ajoute un objet de type `P_Video` au projet et retourne sa clef.	0 - L'ajout ne s'est pas effectué.
`T_ID_MarkPlan addMarkPlan (P_MarkPlan* markplan_)`	Ajoute un objet de type `P_MarkPlan` au projet et retourne sa clef.	0 - L'ajout ne s'est pas effectué.
`T_ID_Homo addHomo (P_Homography* homo_)`	Ajoute un objet de type `P_Homography` au projet et retourne sa clef.	0 - L'ajout ne s'est pas effectué.
`int countMarkPlan ()`	Retourne le nombre d'objets de type `P_MarkPlan` du projet.	-
`int countMarkPlan (T_ID_Plan planid_)`	Retourne le nombre d'objets de type `P_MarkPlan` relié au plan `planid_`.	-
`int countCommonLandmarks (T_ID_Plan planid1_, T_ID_Plan planid2_)`	Retourne le nombre d'objets de type `P_MarkPlan` communs aux plans de clef `planid1_`, `planid2_`.	-
`int suppMap (T_ID_Plan id_)`	Suppression de l'objet de type `P_Map` du projet.	-1 - Objet supprimé, 0 - L'objet n'a pas pu être supprimé.
`int suppMark (T_ID_Mark id_)`	Suppression de l'objet de type `P_Mark` du projet.	-1 - Objet supprimé, 0 - L'objet n'a pas pu être supprimé.
`int suppVideo (T_ID_Plan id_)`	Suppression de l'objet de type `P_Video` du projet.	-1 - Objet supprimé, 0 - L'objet n'a pas pu être supprimé.
`int suppMarkPlan (T_ID_MarkPlan id_)`	Suppression de l'objet de type `P_MarkPlan` du projet.	-1 - Objet supprimé, 0 - L'objet n'a pas pu être supprimé.
`int suppHomo (T_ID_Homo id_)`	Suppression de l'objet de type `P_Homography` du projet.	-1 - Objet supprimé, 0 - L'objet n'a pas pu être supprimé.

`int`
suppDisto
`(T_ID_Disto id_)`
	Suppression de l'objet de type `P_Distortion` du projet.	-1 - Objet supprimé,
		0 - L'objet n'a pas pu être supprimé.

T_ID_Map
`existMap`
`(const char * label_)`
	Vérifie l'existence d'un objet de même type et de même libellé. S'il en existe un, sa clef est retournée.	0 - Il n'existe pas d'autre objet de même libellé.
		-1 - Il existe un objet de même libellé.

T_ID_Video
`existVideo`
`(const char * label_)`
	Vérifie l'existence d'un objet de même type et de même libellé. S'il en existe un, sa clef est retournée.	0 - Il n'existe pas d'objet ayant le même libellé,
		-1 - Il existe un objet de même libellé.

`T_ID_MarkPlan`
`existMarkPlan`
`(T_ID_Mark mark_,`
`T_ID_Plan plan_)`
	Vérifie l'existence d'un objet de même type et de même libellé. S'il en existe un, sa clef est retournée.	-

`T_ID_Homo`
`existHomo`
`(T_ID_Plan id1_,`
`T_ID_Plan id2_)`
	Vérifie l'existence d'un objet de même type et de même libellé. S'il en existe un, sa clef est retournée.	-

`int`
`computeHomo`
`(T_ID_Plan id1_,`
`T_ID_Plan id2_)`
	Calcule l'homographie entre les plans de clef `id1_` et `id2_`.	-1 - L'homographie a pu être calculée,
		0 - Un problème est survenue empêchant le calcul de l'homographie.

`int`
`computeDisto`
`(T_ID_Plan id_)`
	Calcule les paramètres de correction de la distorsion du plan de clef `id_`.	-1 - Les coefficients ont pu être calculés
		0 - Un problème est survenu empêchant le calcul des coefficients

`int`
`exportDescProjet`
`(const char* filename_)`
	Exportation des paramètres du projet en *XML* dans le fichier de nom `filename_`.	-1 - L'export a été effectué
		0 - Un problème est survenu pendant l'export.

`int save ()`	Sauvegarde du projet. Celle-ci est effectuée dans le fichier de nom passé en paramètre du constructeur de `P_Projet` ou du nom renseigné par la méthode `getFilename`.	-1 - La sauvegarde a été effectuée. 0 - Un problème est survenu pendant la sauvegarde du projet.

Méthodes protégées de gestion des clefs

Les méthode de génération de nouvelles clefs d'objets garantissent l'unicité de celles-ci. Elles sont appelées par les méthodes d'ajout de nouvelles références d'objets dans les collections.

Méthodes	Description	Retour d'erreur
`T_ID_Plan nextPlanId ()`	Retourne la clef de type `T_ID_Plan` suivante et incrémente le compteur interne.	0 - Erreur dans la génération de la clef.
`T_ID_Mark nextMarkId ()`	Retourne la clef de type `T_ID_Mark` suivante et incrémente le compteur interne.	0 - Erreur dans la génération de la clef.
`T_ID_MarkPlan nextMarkPlanId ()`	Retourne la clef de type `T_ID_MarkPlan` suivante et incrémente le compteur interne.	0 - Erreur dans la génération de la clef.
`T_ID_Homo nextHomoId ()`	Retourne la clef de type `T_ID_Homo` suivante et incrémente le compteur interne.	0 - Erreur dans la génération de la clef.
`T_ID_Disto nextHomoId ()`	Retourne la clef de type `T_ID_Disto` suivante et incrémente le compteur interne.	0 - Erreur dans la génération de la clef.

5.3.2 P_Mark

Un objet de type `P_Mark` modélise un *landmark* qui peut-être commun à plusieurs plans. Ce sont les objets de type `P_MarkPlan` qui associent les marques aux plans.

Méthodes	Description	Retour d'erreur
`P_Mark (const char* l_)`	Constructeur. Le libellé de la marque est passé en paramètre.	-

`~P_Mark ()`	Destructeur	-
`char*` `getLabel ()`	Retourne le libellé de la marque.	-

int
`setLabel`
`(const char* l_)`
	Modifie le libellé de la marque.	-1 - Le libellé à été modifié.
		0 - Un problème a empêché la modification du libellé.

5.3.3 P_Plan

La classe virtuelle `P_Plan` décrit les propriétés communes aux objets de type `P_Map` et `P_Vidéo`, voir figure 5.14. Le polymorphisme d'héritage permettra dans certains cas de pouvoir considérer l'ensemble des objets de type `P_Map` et `P_Video` comme étant de type `P_Plan`.

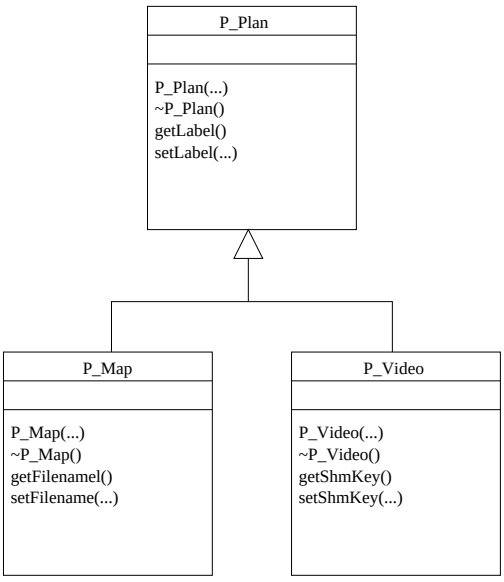


FIG. 5.14 – Graphe d'héritage de P_Plan

Méthodes	Description	Retour d'erreur
`P_Plan` `(const char *label_)`	Constructeur. Le libellé du plan est passé en paramètre.	-
`~P_Plan ()`	Destructeur	-

`const char* getLabel ()`	Retourne le libellé du plan.	-
`int setLabel (const char *l_)`	Modifie de libellé du plan.	-1 - Le libellé a été modifié 0 - Problème lors de la modification.

5.3.4 P_Map

Un objet `P_Map` permet d'utiliser dans l'application des images de cartes. Ces images sont considérées comme des plans. En déterminant les homographies entre les cartes et le repère de référence, il est possible d'effectuer des projections de points dans l'interface graphique afin de vérifier la position de ses *landmarks*.

Méthodes	Description	Retour d'erreur
`P_Map (const char *label_, const char *filename_)`	Constructeur de carte. Le libellé de la carte et le nom de son fichier image sont passés en paramètre.	-
`~P_Map ()`	Destructeur de carte.	-
`const char* getFilename ()`	Retourne de nom du fichier de la carte.	-
`void setFilename (const char *fname_)`	Modifie le nom du fichier de carte.	-

5.3.5 P_Video

Un objet de type `P_Video` gère les données propres à un *buffer* vidéo.

Méthodes	Description	Retour d'erreur

`P_Video (const char *label_, const char *shmKey_, int width = 384, int height = 288)`	Constructeur de plan vidéo. Le paramètre `label_` est le libellé du plan. `shmKey_` est la référence du *buffer* vidéo du *tracker*. Il permet de choisir la caméra à utiliser lorsque plusieurs sont reliées à une même station de travail. `width` et `height` détermine la dimension de l'image. Les valeurs par défaut correspondent à une image demi-PAL.	-
`~P_Video ()`	Destructeur de plan vidéo.	-
`const char * getShmKey ()`	Retourne la référence du *buffer* vidéo.	-
`int setShmKey ( const char *shmKey_)`	Modifie la référence du *buffer* vidéo.	-1 - La référence a été modifiée. 0 - un problème est survenu.
`int getWidth ()`	Retourne la largeur de l'image.	-
`int getHeight ()`	Retourne la hauteur de l'image.	-
`int setDimension (int width_, int height_)`	Fixe les nouvelles dimensions de l'image.	-1 - Les nouvelles dimensions ont été prises en compte. 0 - Erreur

5.3.6 P_MarkPlan

Un objet de type `P_MarkPlan` constitue la relation entre une marque, un plan et les coordonnées de cette marque dans le repère du plan, voir figure 5.15. Il modélise la relation " n - n" entre les objets `P_Mark` et `P_Plan`, c'est-à-dire, qu'une marque peut-être connue dans plusieurs plans et qu'un plan peut avoir plusieurs marques.

Méthodes	Description	Retour d'erreur
`P_MarkPlan (T_ID_Mark idmark_, T_ID_Plan idplan_, T_FloatPoint fp_)`	Constructeur de `P_MarkPlan`. Les clefs du plan et de la marque concernées sont passées en paramètre ainsi que les coordonnées du point dans le repère du plan.	-
`~P_MarkPlan ()`	Destructeur.	-
`T_ID_Mark getMarkId ()`	Retourne la clef de la marque.	-

`T_ID_Plan getPlanId ()`	Retourne le clef du plan.	-
`T_FloatPoint getPoint ()`	Retourne les coordonnées de la marque dans le repère du plan.	
`int setPoint (T_FloatPoint fp_)`	Permet de modifier les coordonnées de la marque dans le repère du plan.	-1 - Le point a été modifié 0 - Un problème est survenu.

5.3.7 P_Line

La classe `P_Line` est une collection de points dans un plan donné. Les coordonnées de ces points sont exprimées dans le repère du plan. Les objets `P_Line` servent à déterminer les paramètres de la distorsion optique des objectifs des caméras et sont utilisés par la classe `P_Distortion`.

Méthodes	Description	Retour d'erreur
`P_Line (T_ID_Plan id_, int increase_ = 3)`	Constructeur. `id_` est la clef du plan dans lequel est la ligne. La collection de points est contenue dans un tableau. Le paramètre optionnel `increase_` permet de spécifier de quelle taille il sera augmenté lorsque cela est nécessaire.	-
`int addPoint (T_FloatPoint fp_)`	Ajout d'un point supplémentaire à la ligne.	-1 - Le point a été ajouté. 0 - Il y a une erreur.
`T_ID_MarkPlan* getPlanId()`	Retourne la clef du plan.	-
`T_FloatPoint getPoint(int i_)`	Retourne le i^{eme}point de la ligne.	-
`int count ()`	Retourne le nombre de points.	-

5.3.8 P_Homography

Un objet de la classe `P_Homography` est instancié lors du calcul d'une homographie entre deux plans. Il n'accède pas directement à la bibliothèque externe de calcul, il utilise un objet de type `P_HomoCalc` de liaison, voir figure 5.16. En cas de changement de bibliothèque de calcul,

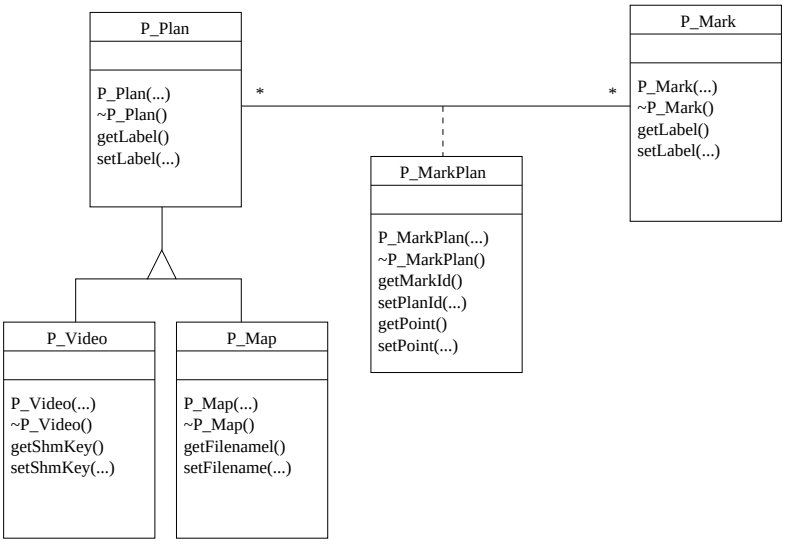


FIG. 5.15 – Liens n-n entre les marques et les plans

ce sont les classes du module de liaison qu'il faudrait réécrire sans toucher à `ParkviewApp`.

Méthodes	Description	Retour d'erreur
`P_Homography (T_ID_Plan planSrc_, T_ID_Plan planDest_)`	Constructeur. Les clefs des plans source et de destination sont passées en paramètre.	-
`~P_Homography ()`	Destructeur	-
`T_ID_Plan getPlanSrc ()`	Retourne la clef du plan source.	-
`T_ID_Plan getPlanDest ()`	Retourne la clef du plan destination.	-
`int addCorresPoints (T_FloatPoint psrc_, T_FloatPoint pdest_)`	Ajoute un couple de points avec `psrc_` dans le repère du plan source et `pdest_` dans le repère du plan destination.	-1 - Le couple a été ajouté, 0 - Le couple n'a pas pu être ajouté.
`int computeHomo ()`	Lance le calcul de l'homographie. Il faut un minimum de quatre couples de points pour appeler cette méthode.	-1 - L'homographie a été calculée, 0 - Erreur.
`int reset ()`	Supprime tous les couples de points.	-1 - Les correspondances ont été supprimées, 0 - Erreur.
`const T_Matrix33 getMatrix ()`	Retourne un tableau contenant les coefficients de la matrice d'homographie.	-

`T_FloatPoint calcPoint (T_FloatPoint fp_)`	Retourne l'image du point `fp_` par l'homographie.	-

Calcul d'homographie en utilisant la bibliothèque *Gandalf.*

La classe `P_HomoCalc` fait appel à la bibliothèque *Gandalf* pour le calcul des homographies. Cette bibliothèque propose un ensemble de fonctions et de structures de données tel que décrit dans [McL99].

Dans un premier temps, il faut définir une structure de données devant recevoir les correspondances entre les points.

```
Gan_SymMatEigenStruct SymEigen;
/* initialise eigensystem matrix */
gan_homog33_init ( &SymEigen );
```

Ensuite, la fonction `gan_homog33_increment_p` permet d'ajouter des correspondances au système d'équations géré par la bibliothèque :

```
int iEqCount=0, iCount;
/* p1 et p2 sont les tableaux de points de correspondance
   dans les deux plans.*/
Gan_Vector3 p1[NBCORRESP], p2[NBCORRESP];
for ( iCount = 0; iCount < NBCORRESP; iCount++ ){
  gan_homog33_increment_p (&SymEigen, &p1[iCount],
                           &p2[iCount], 1.0, &iEqCount );
}
```

L'argument, `iEqCount`, est le compteur de l'ensemble des équations introduites dans le système.

Une fois que l'ensemble des points de correspondance est introduit dans le système, nous pouvons le résoudre en utilisant :

```
Gan_Matrix33 m33P; /* homography matrix P */
gan_homog33_solve ( &SymEigen, iEqCount, &m33P );
```

Il est possible de répéter le calcul de l'homographie avec de nouvelles données en utilisant :

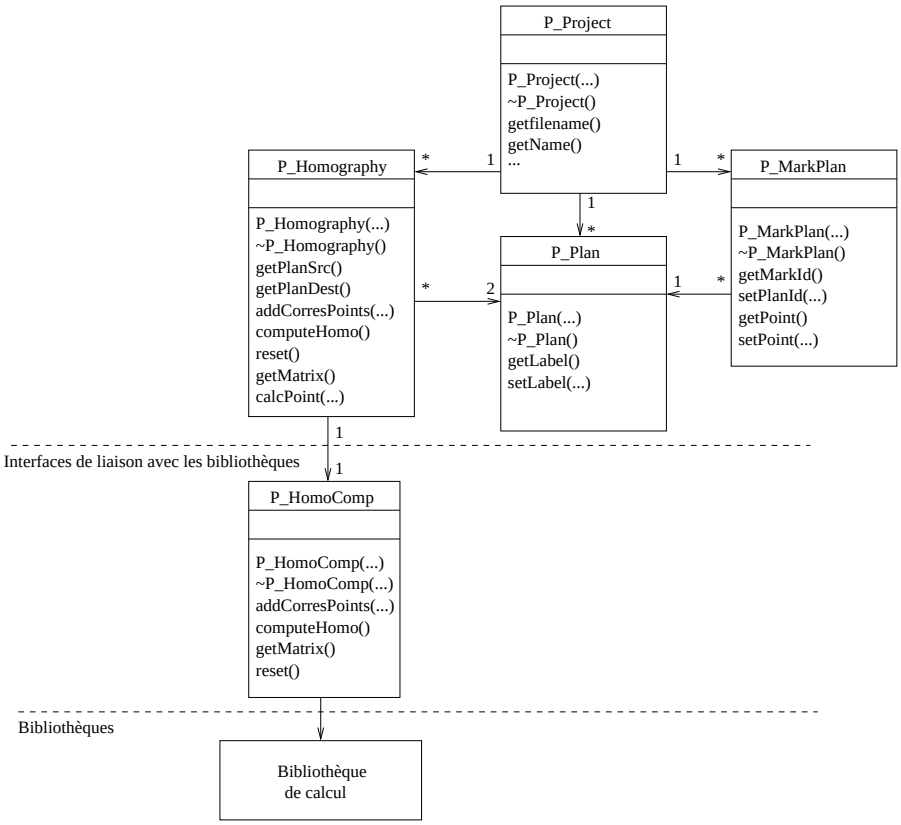


FIG. 5.16 – Diagramme *UML* des classes de calcul homographique

```
gan_homog33_reset ( &SymEigen );
```

A la fin du calcul, il est possible de vider les structures de données en utilisant la fonction :

```
gan_homog33_free ( &SymEigen );
```

5.3.9 P_Distortion

Un objet de la classe `P_Distortion` est instancié lors du calcul des coefficients de distorsion d'un plan. Cet objet n'accède pas directement aux bibliothèques externes de calcul, il utilise un objet de type `P_DistoCalc` de liaison dont la classe est défini dans le module de liaison aux bibliothèques.

Méthodes	Description	Retour d'erreur
`P_Distortion (T_ID_Plan planid_)`	Constructeur. La clef du plan est passée en paramètre.	-
`~P_Distortion ()`	Destructeur.	-
`T_ID_Plan getPlan()`	Retourne le clef du plan.	-
`int addLine (P_Line* line)`	Ajoute la connaissance d'une nouvelle ligne dans l'image.	-1 - La ligne a été ajoutée, 0 - Erreur.
`int computeDisto ()`	Calcule les coefficients de distorsion.	-1 - Les coefficients ont été calculés, 0 - Erreur.
`int reset ()`	Supprime toutes les lignes.	-1 - Les lignes ont été supprimées, 0 - Erreur.
`T_FloatPoint calcPoint (T_FloatPoint fp_)`	Retourne les coordonnées du point `fp_` corrigées.	-
`T_CoefDisto getCoef ()`	Retourne les coefficients de distorsion.	-

5.4 Le *tracker Prima*

Le système de suivi de l'équipe *Prima* est ce que l'on appelle un processus récursif d'estimation. Celui-ci intègre des informations en provenance des images. Le temps de cycle est limité

par leur processus d'acquisition à $\frac{1}{25}^{eme}$ de seconde. L'approche de *Prima* consiste à exploiter les prédictions issues du processus d'estimation réalisé par un filtre de Kalman pour focaliser les traitements à une partie restreinte de l'image. À partir des prédictions, les traitements sont appliqués d'une manière locale à chaque pixel afin d'estimer la possibilité que l'objet suivi se trouve à ce pixel. Ces estimations sont ensuite combinées d'une manière statistique, d'abord par suppression des points externes par pondération avec une fenêtre Gaussienne, et ensuite par estimation des moments pour déterminer la confiance, la position et l'envergure d'une détection.

Afin de garantir la robustesse de son suivi, le *tracker* peut combiner la détection des cibles par la différence d'image de fond, par mouvement et par couleur. Ces trois techniques de détection sont utilisées par un processus récursif de suivi. Le système est contrôlé par un superviseur d'exécution capable d'adapter les traitements afin d'assurer un cycle à la vitesse vidéo, voir figure 5.17.

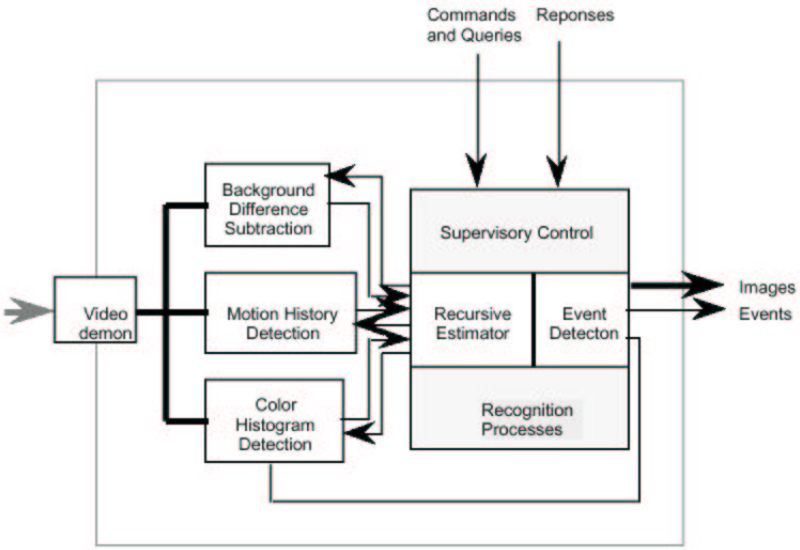


FIG. 5.17 – Architecture du *tracker Prima*

5.4.1 Le superviseur

Le premier rôle du superviseur est d'accepter et d'interpréter les commandes, incluant les interrogations sur l'état du processus, et les demandes de description des capacités de traitement. Le deuxième rôle du superviseur est d'assurer la gestion des paramètres afin d'assurer les contraintes fixées par les commandes. Par exemple, le contrôleur du capteur regarde le temps de calcul pour chaque cycle. Par défaut, un superviseur assure un cycle de suivi avec priorité

à la vitesse du cycle et une deuxième priorité au nombre de cibles. S'il devient impossible d'assurer le cycle de traitement à une vitesse vidéo, le superviseur peut réduire la précision (en traitant un pixel sur N) ou il peut réduire le nombre de cibles, ce qui n'est pas souhaitable dans notre cas. Heureusement, la gestion de ces priorités peut être paramétrée par commande au superviseur. Le *tracker* est prévu pour être embarqué dans un noyau *CORBA*, permettant la communication de ces résultats avec d'autres processus au travers des communications avec *UDP* ou *TCP/IP*. Cette possibilité sera utilisée dans les évolutions ultérieures de *ParkView*.

Dans notre cas, nous ne combinerons que les modules de détection par différence de fond et par mouvement appelés respectivement *background tracker* et *motion tracker*. En effet, nous ne pouvons connaître à l'avance les couleurs des objets qui se présentent sur le parking. Néanmoins, nous verrons dans la partie 6.2.2 comment il a été envisagé d'utiliser le suivi par couleur, *color tracker*, pour déterminer l'orientation de notre robot mobile.

5.4.2 Mise en oeuvre

L'ensemble des modules du *tracker* est réalisé en *C++*. Sa mise en oeuvre est faite par programmation événementielle. Elle consiste à définir une action à effectuer pour chaque événement relevé par le *tracker*. Pour cela, un cadre de programmation existe sous la forme de la classe virtuelle `Supervisor` fournissant la signature de l'ensemble des méthodes à implémenter. Il en existe une par événement différent. Il suffit donc de définir une classe dérivée de `Supervisor` est d'implémenter ses méthodes en fonction des actions que l'on désire associer à chaque événement. Cette classe sera alors instanciée et utilisée par le *tracker*. Les méthodes définies dans `Supervisor` sont :

```
virtual void eventObservRegion(const RobustTarget &rt,        // Cible
                               const TriggerRegion & region,  // Region de détection
                               int id ,                       //
                               observEvent Case) = 0 ;        //
virtual void eventExitRegion(RobustTarget & rt,
                             const TriggerRegion & region,
                             int id) = 0 ;
virtual void eventFaceDetection(const RobustTarget &rt) = 0 ;
virtual void eventMoveTarget(const RobustTarget &rt) = 0;
virtual void eventNewTarget(const RobustTarget &rt) = 0;
virtual void eventSplitTarget(const RobustTarget &rt0,
                              const RobustTarget &rt1,
                              const RobustTarget &rtParent) = 0;

virtual void eventSplitOffTarget(const RobustTarget &rt,
```

```
                                    int parentId) = 0;

virtual void eventMergeTarget(const RobustTarget &rt,
                              const RobustTarget &rtParent0,
                              const RobustTarget &rtParent1) = 0;
virtual void eventDeleteTarget(const RobustTarget &rt) = 0;
virtual void eventDeleteTargetManually(const RobustTarget &rt) = 0;
```

5.4.3 Paramétrage

Le paramétrage du *tracker* utilise un fichier de configuration *XML* décomposé en 4 parties :

Général Les paramètres généraux sont communs à l'ensemble des modules utilisés et concernent plus particulièrement leur gestion. On peut y trouver : les paramètres du *filtre de Kalman*, la gestion des fusions et séparations de cible (*split/merge*), les modules à utiliser correspondant aux différents types de suivi (*color*, *background*, *motion*).

Contrôle Les paramètres de contrôle s'intéressent à la gestion des cibles et concernent la robustesse de leur suivi. Ainsi, si une cible a été vue durant un nombre de *frames* (images) supérieur à un seuil défini dans cette partie, elle est considérée robuste. Les régions d'observation telles que les zones d'entrée, de sortie et d'exclusion sont également définies à cet endroit. Cette partie gère également les références des *buffers* `shmkey` en permettant de sélectionner la caméra à utiliser lorsque plusieurs sont reliées à la machine.

Module Cette partie concerne les paramètres génériques aux différents modules utilisables. Les paramètres sont définis de façon globale à tous les modules de suivi. On pourra cependant les redéfinir pour un module particulier dans la partie "tracker" décrite ci-dessous.

Tracker Cette partie concerne les paramètres propres à un module de suivi. Il peut s'agir de la définition de l'histogramme de couleur dans le cas du *colortracker*, du nombre d'images de fond à intégrer dans le cas du *backgroundtracker*. Cette partie permet aussi de définir des seuils tels que la taille minimum des cibles suivies ou encore l'équilibre à prendre en compte entre les observations et les estimations faites par le filtre de Kalman. Le nombre et la nature des paramètres de cette partie évoluent en fonction des modules qui sont ajoutés à l'architecture générale.

5.5 Le serveur de carte `ParkviewMapServ`

Le serveur de carte *ParkviewMapServ* permet aux clients *ParkView* de pouvoir récupérer la dernière carte des positions des objets. Il est architecturé autour d'entités logicielles asynchrones, voir figure 5.18. Ceci permet aux différents processus d'avoir des cycles de traitement de longueur variable. C'est le cas, par exemple, du *tracker Prima* qui adapte la longueur de son cycle en fonction du nombre et de la complexité des cibles suivies.

Cette première version de *ParkviewMapServ* fonctionne sur une seule machine. Les divers *processus* et *thread* utilisent des segments de mémoire partagée. Pour les *trackers*, qui sont des processus indépendants, cet accès utilise les pointeurs de segment basés sur les descripteurs de fichier du système *Linux*.

Dans notre architecture, les *flags* partagés permettent aux divers processus de vérifier si les données en amont ont été modifiées depuis leur dernière lecture. Si ce n'est pas le cas, la lecture se fait alors dans un cache mémoire au sein du processus demandeur sans monopoliser le segment de mémoire partagée. Sur la figure 5.18, les dispositifs de gestion de la concurrence d'accès à la mémoire ne sont pas représentés.

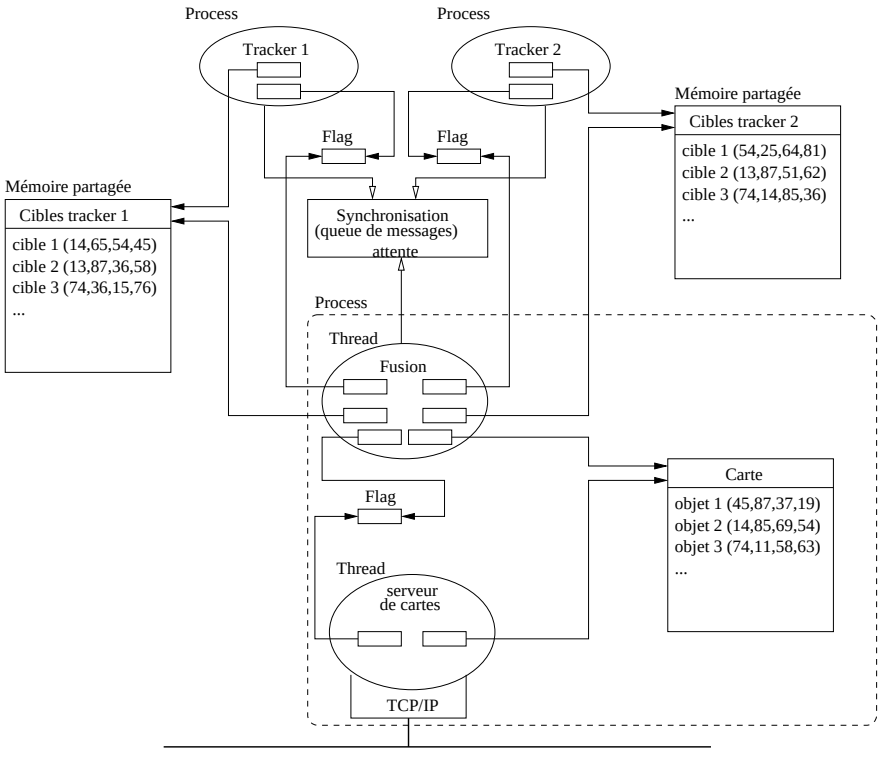


FIG. 5.18 – Architecture générale du serveur de cartes

5.5.1 Les processus *tracker*

Les processus *tracker* utilisent un objet de type `P_Tracker` qui est une classe dérivée de `Supervisor`, voir figure 5.19. Lorsque les caractéristiques d'une cible suivie par le *tracker* changent, la méthode `eventMoveTarget` est appelée avec l'ensemble des données sur la cible en paramètre. La mise à jour de la liste des cibles est effectuée en appelant la méthode `updateTarget` de l'objet `P_ShareTargetList` gérant la mémoire partagée. Nous avons défini une classe générique de gestion de la mémoire partagée `P_MemShare` utilisée par l'ensemble des traitements de *ParkviewMapServ*.

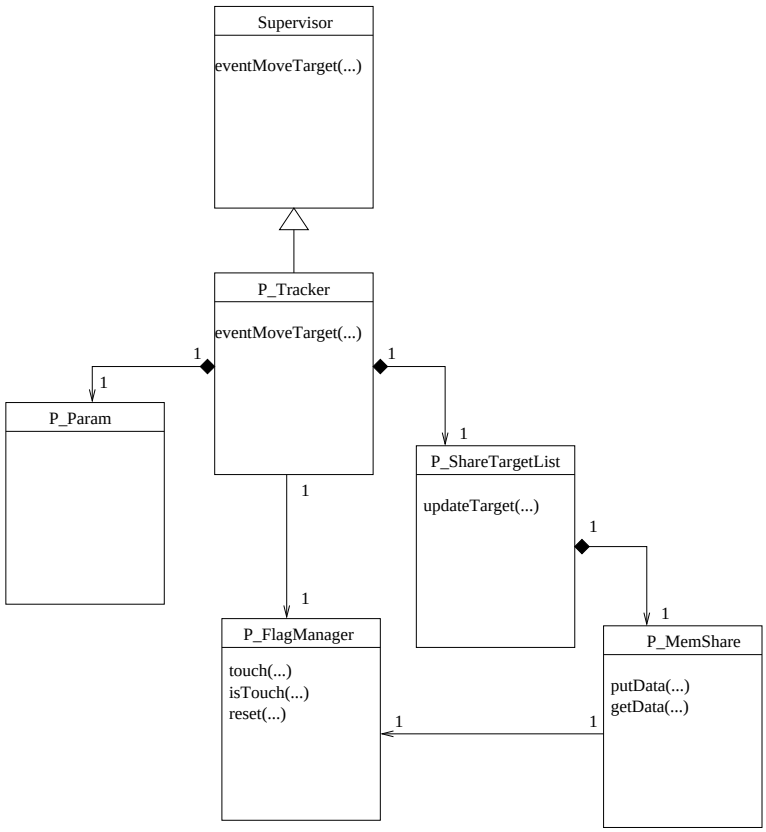


FIG. 5.19 – Graphe de classes *UML* du processus *tracker* de *ParkviewMapServ*

5.5.2 Le processus serveur

Le processus serveur effectue à la fois les tâches de fusion de données et de serveur de cartes. Ces deux fonctions sont asynchrones et exécutées en parallèle sous forme de *thread*. Ceci permet de bénéficier d'une indépendance de fonctionnement tout en partageant des objets communs, voir figure 5.20.

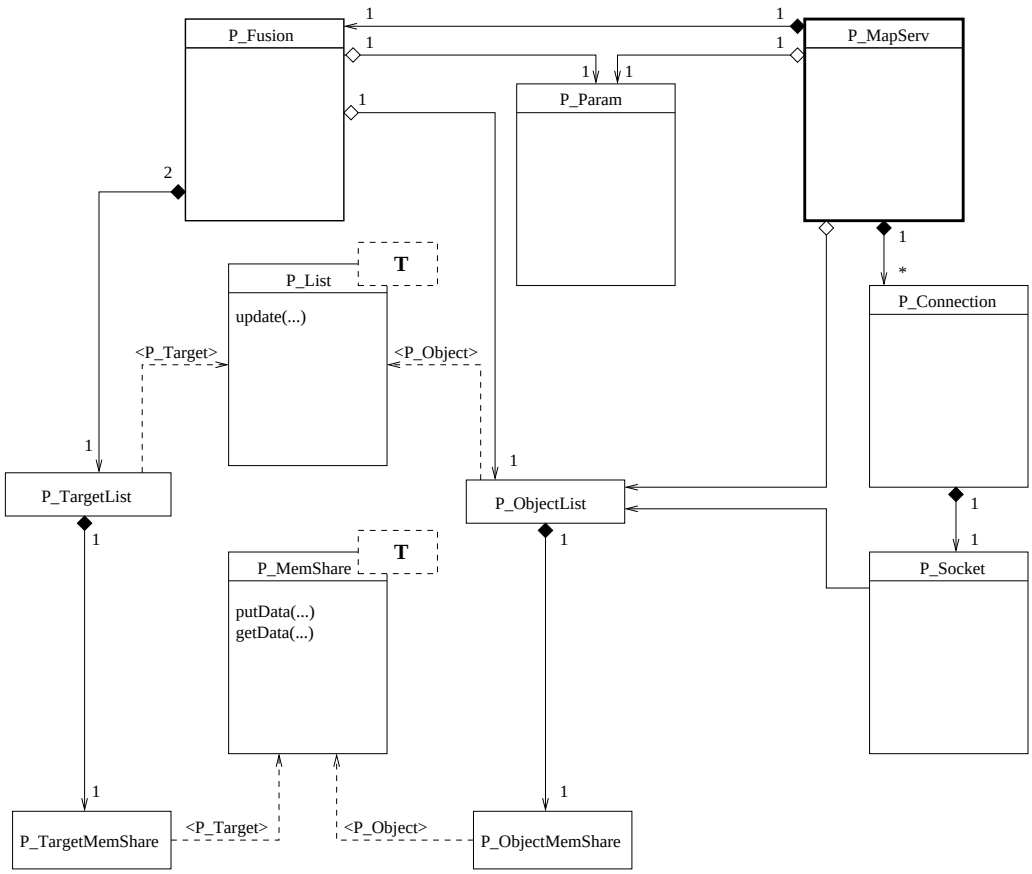


FIG. 5.20 – Graphe de classes *UML* du serveur de cartes

Le *thread* de fusion est en attente de déclenchement par un des *trackers* ayant remis à jour sa liste de cibles. Ceci est assuré par un mécanisme de synchronisation fourni par le système d'exploitation (queue de messages) représenté dans le diagramme général 5.18. La fusion est réalisée à partir des deux listes de cibles des *trackers* gérées par des objets de type `P_TargeList,` suivant l'algorithme décrit en 3.4.2. Le résultat est disponible pour le *thread* serveur sous la forme d'un segment de mémoire partagée géré par l'objet de type `P_ObjectList`. On notera la présence des classes de gestion de segments de mémoire partagée et de listes génériques.

Le serveur de cartes commence par ouvrir un *socket* serveur en attente de connexion. Lors d'une demande de carte, celle-ci est prise en compte par un *thread* de connexion utilisant un objet de type `P_Connection`. Celui-ci vient alors interroger l'objet partagé `P_ObjectList` contenant la dernière carte à jour.

5.6 Test de fusion de données

Nous avons effectué des tests de fusion de données sur la banc de test décrit en 4.2. Ils ont été réalisés à l'aide d'une balle de tennis venant percuter de petits obstacles pour dévier sa trajectoire.

La figure 5.21 montre la balle visible par les deux caméras. Les observations des *trackers* apparaissent en sur-impression vidéo. On peut également remarquer les "zones d'entrée" des *trackers* délimitées par les rectangles de chaque côté des images.

Dans cette exemple, nous avons utilisé le *colortracker* qui a été paramétré pour ne détecter que le jaune de notre balle. Dans la partie inférieure de la figure, nous voyons les données brutes en sortie de chaque *tracker*.

Après avoir calculé les paramètres des transformations depuis les plans vidéo vers le repère de référence, nous avons effectué la fusion de données. Les résultats sont reportés sur le dessin du plan du banc de test. Dans la figure 5.22, nous avons coloré :

- en rouge, la trajectoire détectée par le *tracker* 1,
- en bleu, celle du *tracker* 2,
- en jaune, la fusion des trajectoires lorsque l'objet suivi est composé de deux cibles.

La figure 5.23 met en évidence la trajectoire issue de la fusion des données. Nous pouvons remarquer qu'elle se situe à mi-distance des deux mesures des *trackers* conformément à la fonction fus définie en 3.4.1.

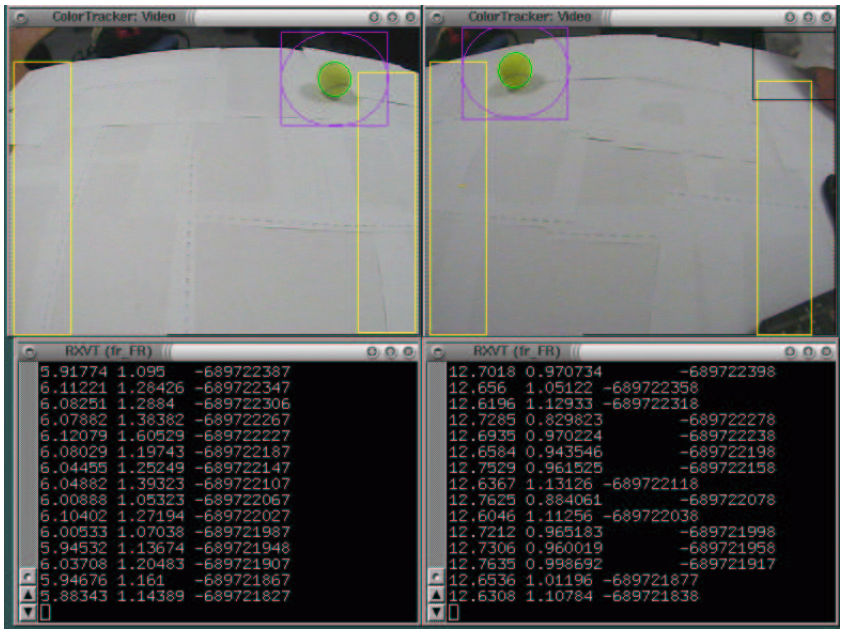


FIG. 5.21 – Suivi d'une balle de tennis par deux *trackers*

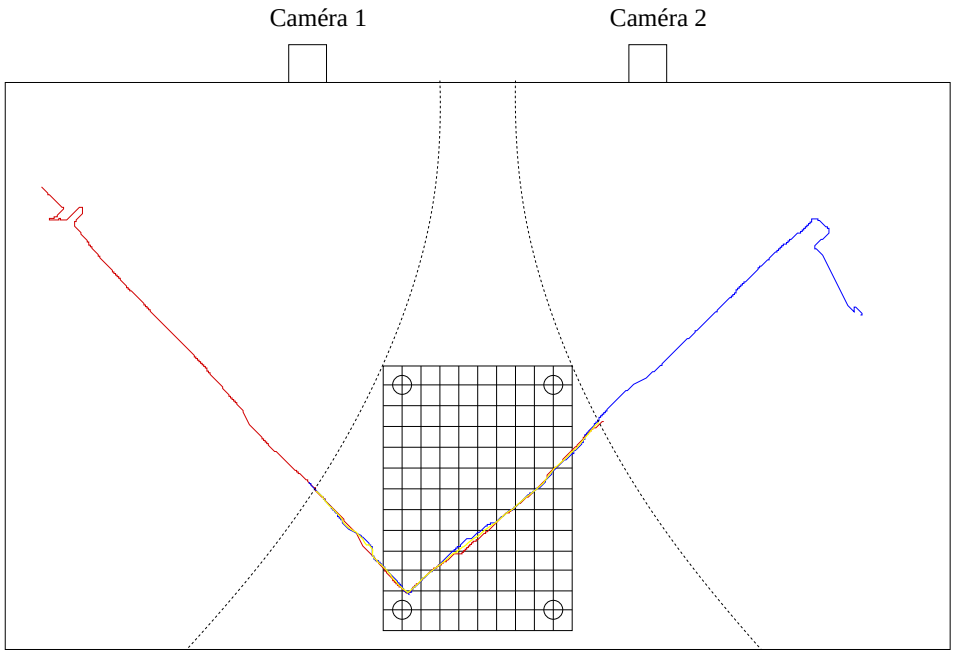


FIG. 5.22 – Représentation de la trajectoire de la balle suivie par deux *trackers*

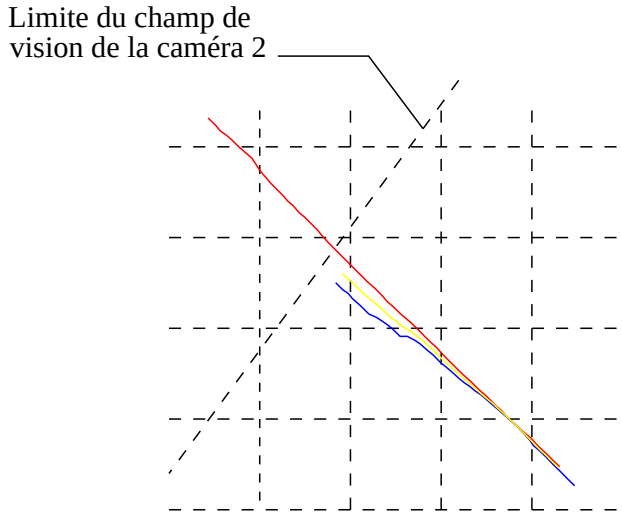


FIG. 5.23 – Représentation détaillée de la trajectoire de la balle suivie par deux *trackers*

Chapitre 6

Applications

Dans ce chapitre, nous décrivons les développements effectués dans le cadre de deux applications utilisant l'infrastructure *ParkView*.

La première concerne la mesure de vitesse automatisée sur route. Nous avons mis en pratique l'interface de paramétrage de ParkView et développé un module particulier de mesure.

La seconde est un expérimentation concernant l'évitement d'obstacles mobiles pour le *Cycab*. Celle-ci a permis d'éprouver la modularité et l'extensibilité de notre infrastructure en lui adjoignant un capteur laser.

6.1 La mesure de vitesse

L'équipe *Prima*, par l'intermédiaire de la société *Blue Eye Video*, développe des produits de mesure destinés au relevé de vitesse sur route. L'une des premières expérimentations de l'application *ParkView* a été le paramétrage d'une expérimentation menée en condition réelle.

6.1.1 Contexte expérimental

La figure 6.1 présente le contexte expérimental. Une caméra vidéo est placée sur un pont surplombant une autoroute. Aucun paramètre expérimental n'est fixe d'une manipulation sur l'autre. Ainsi, pour une autre mesure, la hauteur du pont pourra être différente, l'angle de prise de vue également. De la même manière, la caméra pourrait très bien être placée au sommet d'un mat en bord d'autoroute. La mise en place du système de mesure doit être extrêmement souple. Elle sera réalisée, à terme, par des personnes sans connaissance particulière en optique.

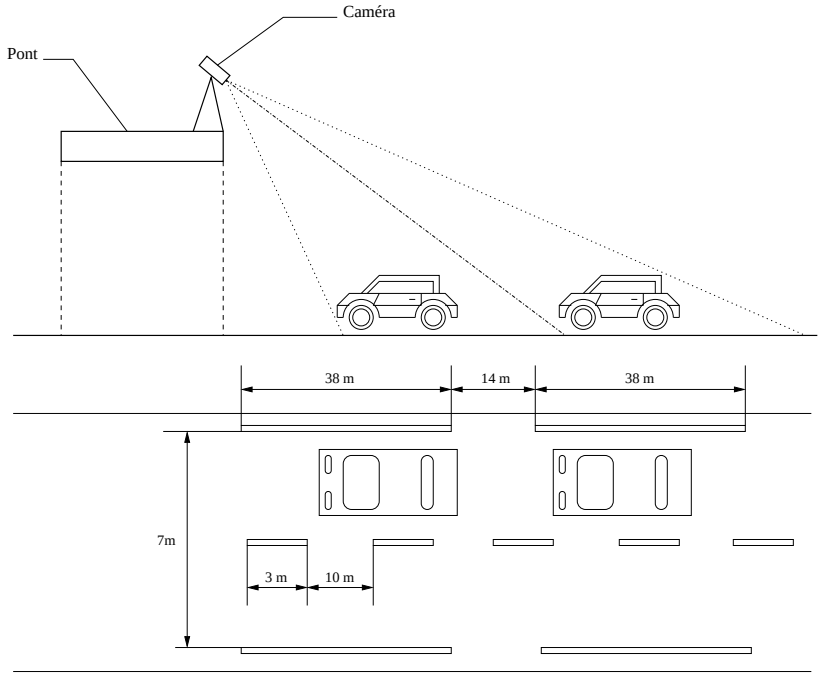


FIG. 6.1 – Mesure de la vitesse sur route

6.1.2 Architecture de l'application

Pour cette application, il est possible de créer un client au serveur de cartes *ParkviewMapServ*. Celui-ci met à disposition des cartes de véhicules dans un repère coplanaire à l'autoroute. Cette architecture est décrite en figure 6.2. On notera que, sur ce diagramme, le client cinémomètre est également relié à la caméra vidéo. Cela est nécessaire pour pouvoir faire l'acquisition de séquences vidéo lorsqu'un véhicule est en excès de vitesse. Néanmoins, il n'y a pas, à ce niveau, de traitement sur l'image. A terme, le système devra être relié à une deuxième caméra "gros plan", pour pouvoir effectuer l'identification des véhicules.

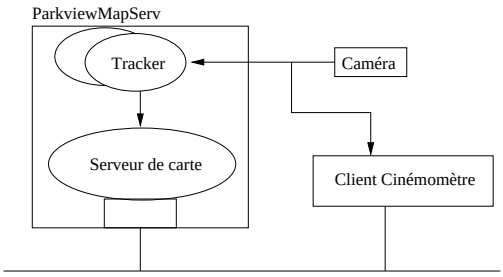


FIG. 6.2 – Utilisation d'un client cinémomètre

Dans notre cas, il n'est pas nécessaire de disposer des fonctions de fusion de *ParkviewMap-*

Serv puisque cette application fonctionne avec une seule caméra. Nous pouvons donc travailler directement avec le système de cibles du *tracker*. Le figure 6.3 montre le diagramme d'instances utilisé pour cette expérimentation.

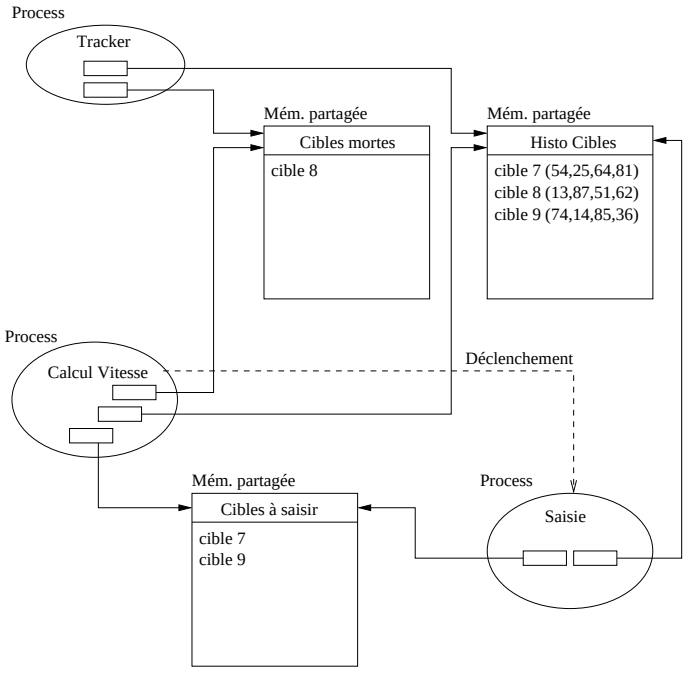


FIG. 6.3 – Architecture de l'application de mesure de vitesse

Cette architecture utilise deux processus asynchrones fonctionnant en parallèle. Le processus de suivi met à jour des tables dans un segment de mémoire partagée. Une première table contient l'historique des informations sur les cibles en cours de suivi. Cette table est utilisée par le processus de calcul de vitesse pour déterminer au mieux la vitesse des cibles en intégrant pour chacune d'elles l'ensemble de ses mesures. Lorsque le processus de *tracking* perd une cible, ce qui se produit quand elle sort du champ de vision de la caméra, il inscrit son numéro de référence dans la table des cibles mortes. C'est le processus de calcul de vitesse qui se chargera de purger la table d'historique en fonction de ces informations. Il purgera ensuite la table des cibles mortes. Le processus de calcul de vitesse a la possibilité, en utilisant les primitives de synchronisation du système (queues de messages), de réveiller le processus de saisie. C'est le cas lorsque la vitesse d'une cible dépasse un seuil fixé en paramètre. Celui-ci vient alors lire la liste des cibles à saisir et évalue, à l'aide de leur historique, le meilleur moyen de le faire (position estimée, meilleur moment de passage devant la caméra "gros plan", lancement du module externe *OCR* de lecture des plaques d'immatriculation des véhicules, etc...). Il purge ensuite la table des cibles à saisir. Lorsqu'une cible est signalée comme à saisir et, en même temps, comme morte, le processus de calcul de vitesse attend que le processus de saisie

l'ait supprimée de la liste des cibles à saisir avant de supprimer à son tour son historique.

La figure 6.4 présente le diagramme *UML* d'arborescence des classes de l'application.

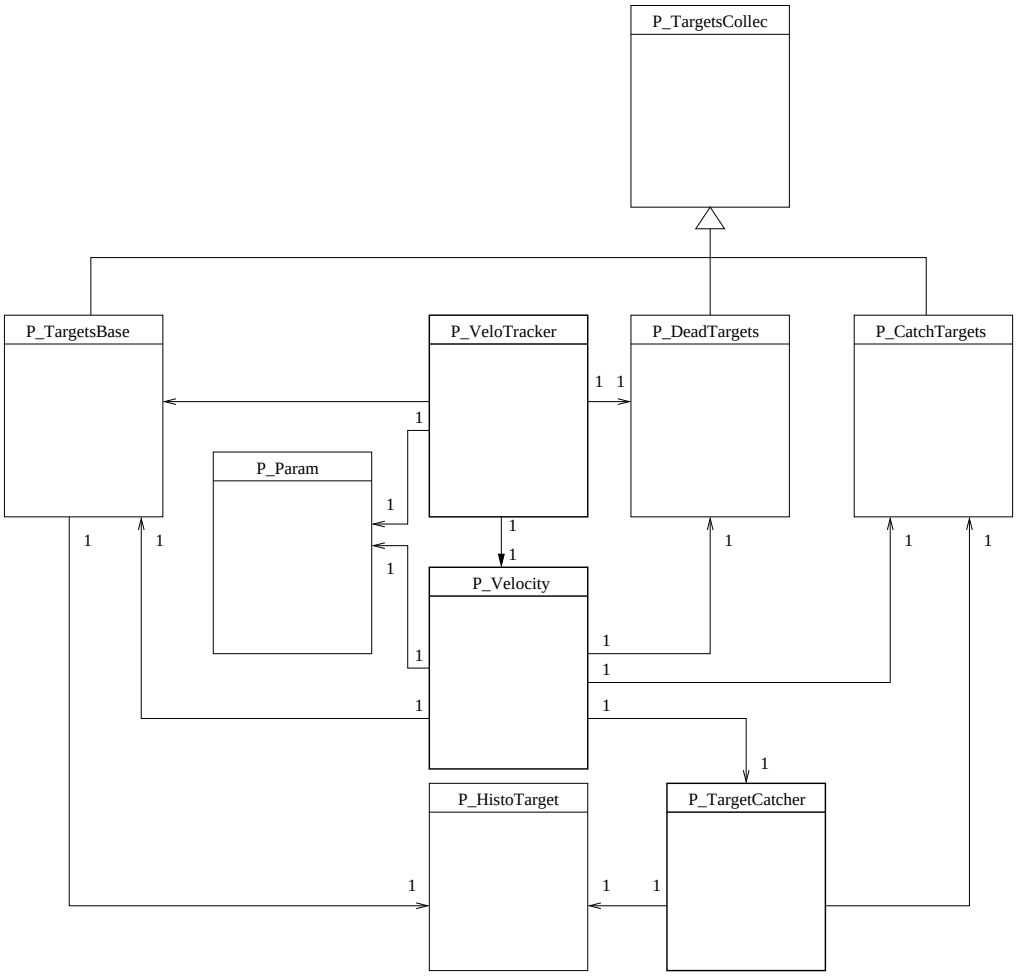


FIG. 6.4 – Diagramme UML de l'application de mesure de vitesse

La classe `P_Velocity` implémente le processus de calcul de vitesse. Celle-ci accède aux trois objets de gestion des listes de cibles suivies, cibles mortes et cibles à saisir qui sont respectivement de type `P_TargetsBase`, `P_DeadTargets` et `P_CatchTargets`. Ces classes sont dérivées de la classe de gestion de liste de cibles `P_TargetsCollec`. L'objet `P_TargetsBase` instancie un objet de type `P_HistoTarget` pour chaque cible suivie. Il gère l'historique des évènements pour chaque cible suivie.

Le processus *tracker* est implémenté par `P_VeloTracker`. Il accède, via les primitives de concurrence d'accès du système, à la base des cibles suivies et purge la liste des cibles mortes. De la même manière, `P_TargetCatcher` implémente le processus de saisie de cibles.

6.1.3 Détermination de la vitesse

Le calcul de vitesse s'effectue par dérivation de la position de la cible, dans le plan de la route, par rapport au temps. Ainsi, pour chaque trame vidéo, soit tous les $\frac{1}{25}^{eme}$ de seconde, nous connaissons la position de la cible suivie dans le plan de la route. En calculant la distance parcourue par différence, nous calculons une mesure de vitesse pour chaque trame. Le passage d'un véhicule génère environ 50 mesures de vitesse. Afin de déterminer la vitesse qui sera retenue comme étant celle de notre véhicule, nous effectuons un filtrage gaussien dynamique. Pour cela, nous faisons une moyenne du premier tiers des vitesses mesurées, pondérée par une gaussienne centrée sur une valeur fixe. Celle-ci peut être, par exemple, la limitation de vitesse de la voie observée. Nous calculons ensuite la moyenne de l'ensemble des vitesses pondérées par une gaussienne centrée autour de la première vitesse moyenne calculée, voir figure 6.5.

Ce filtrage dynamique permet d'absorber les imprécisions en sous-pondérant les vitesses trop éloignées de l'ensemble.

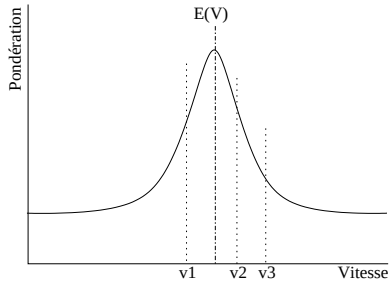


FIG. 6.5 – Filtrage gaussien dynamique des vitesses

6.1.4 Paramétrage

La figure 6.6 présente une image de l'autoroute telle qu'elle apparaît dans l'interface de paramétrage de plan vidéo *ParkView*.

FIG. 6.6 – Interface de paramétrage *ParkView*

La première étape consiste à marquer les points particuliers dans l'image. Dans notre cas, nous connaissons les dimensions exactes des marquages au sol. Il nous faut, au minimum, quatre points pour calculer les paramètres de l'homographie entre le plan vidéo et le plan de référence confondu à la route. La figure 6.7 montre les *landmarks* utilisés pour cette ex-

FIG. 6.7 – *Landmarks* utilisés

périmentation. Ces *landmarks* sont instanciés dans le plan de référence, la route, avec leurs coordonnées en mètre. La figure 6.8 montre la liste des libellés utilisés. On génère ensuite le

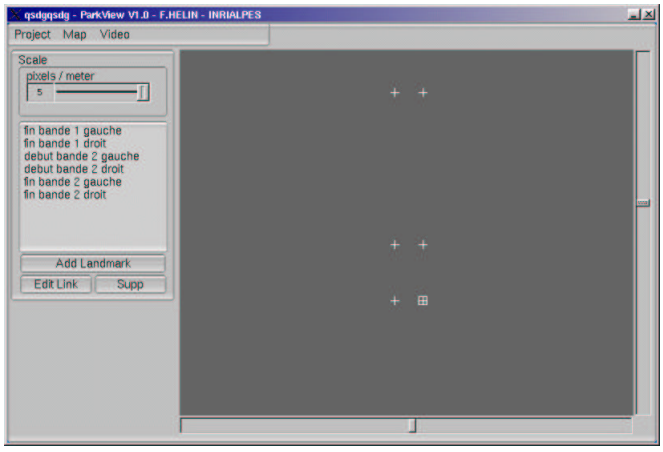


FIG. 6.8 – Plan de référence - Route

fichier de paramètres pour le module d'exploitation dont un exemple est donné ci-après.

```
### --- fichier de paramétrage ---
<video id = 2 >
  <label value = "camera autoroute"/>
  <shmkey value = "0"/>
  <homography>
    <m11 value =0.808673/>
    <m12 value =0.291428/>
    <m13 value =-115.111/>
    <m21 value =0.218871/>
    <m22 value =-0.292512/>
    <m23 value =547.21/>
    <m31 value =0.0018842/>
    <m32 value =0.0977101/>
    <m33 value =1/>
  </homography>
</video>
```

6.1.5 Résultats

Le figure 6.9 montre l'écran de visualisation de l'application de mesure de vitesse . Dans

FIG. 6.9 – Ecran de visualisation d'une session de mesures de vitesse

la partie inférieure droite, les zones de délimitation d'entrée, de sortie et d'expulsion sont affichées en sur-impression sur l'image vidéo. La bande supérieure est gérée par le processus de saisie qui affiche, sous forme de mosaïque, les images des véhicules dépassant le seuil de vitesse fixé en paramètre. L'ensemble des vitesses mesurées par le module de calcul de vitesse est affiché en bas à gauche. A chaque référence de véhicule est associée sa vitesse.

6.2 L'application d'évitement d'obstacles

L'évitement d'obstacles en milieu dynamique est un des sujets de recherche de l'équipe *CyberMove*. Les obstacles considérés dans cette application peuvent être des robots mobiles de type *Cycab*, des voitures se déplaçant sur le parking ou encore des piétons.

Le travail réalisé ici consiste à mettre à disposition du module d'évitement d'obstacles l'ensemble des informations dont il a besoin pour réaliser sa tâche. Nous nous intéresserons ainsi à la façon de déterminer l'orientation θ de notre robot mobile ainsi qu'à la manière de le discriminer des autres objets mobiles sur le parking. Nous présenterons ensuite l'architecture cliente embarquée *ParkView*.

6.2.1 Les "non-linear velocity obstacles"

L'approche des *"non-linear Velocity Obstacles"* consiste à déterminer à chaque rafraîchissement de la carte des obstacles mobiles leur vitesse permettant d'éviter l'ensemble des obstacles sur une période de temps donnée, en considérant alors que ces derniers suivent toujours les trajectoires prédites pendant la période de temps considérée. La nouveauté de cette approche vient du fait qu'elle travaille directement en degré 1, c'est-à-dire, sur les vitesses du véhicule et des obstacles mobiles. Il en résulte des constructions d'obstacles directement dans l'espace des vitesses, avec des expressions analytiques relativement simples.

Le figure 6.10 est une représentation graphique des *NLVO*. Les obstacles mobiles sont ici assimilés à des disques (*bounding box*). Les parties non grisées représentent l'espace des vitesses admissibles du robot mobile piloté (i.e. les vitesses n'engendrant pas de collisions avec les obstacles) à partir de l'évolution des vitesses instantanées des obstacles mobiles. Le champ d'application de cette approche n'est pas limité aux véhicules mobiles, il présente un intérêt évident pour d'autres applications réelles ou virtuelles (e.g. avions, bateaux, jeux vidéo. . .). Ce travail de recherche est présenté dans [LSSL02a, LSSL02b]

L'approche des *NLVO* demande à connaître la configuration du robot à chaque instant. Celle-ci, représentée par un vecteur $\begin{pmatrix} x & y & \theta \end{pmatrix}^T$, correspond à sa localisation dans le repère considéré ainsi que son orientation. Dans les informations que nous fournit le serveur de cartes, nous n'avons pas l'information sur l'orientation du robot. Il nous faut donc mettre en place un système capable de nous renseigner sur θ. D'autre part, parmi les objets suivis par *ParkView*, il nous faut déterminer lequel correspond au robot mobile piloté.

6.2.2 Détermination de θ par pastilles de couleur

Afin de déterminer l'orientation de notre robot mobile, une première voie explorée a été d'utiliser deux *colortrackers* fonctionnant en parallèle.

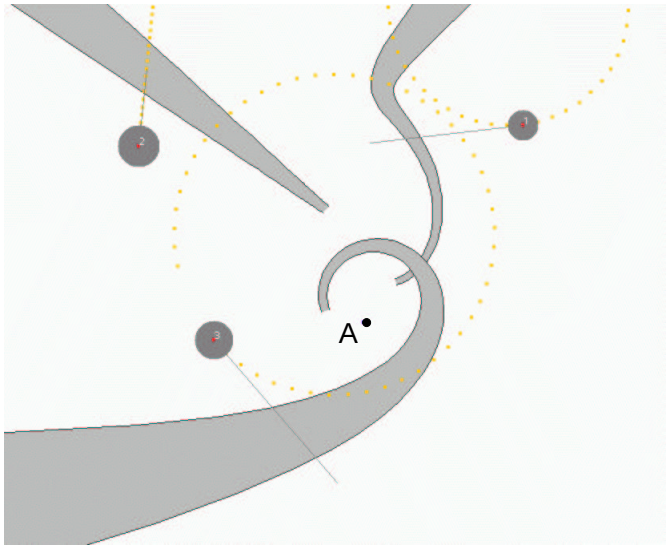


FIG. 6.10 – Représentation graphique des *non-linear velocity obstacles*

L'approche

Cette approche consiste à équiper le robot *Cycab* de deux pastilles de couleur sur son toit. Chacune d'elle est détectée par un *tracker* dont le paramétrage a été fait sur sa couleur. La figure 6.11 montre le modèle réduit commandé à distance et les cibles détectées par les deux *colortrackers* alors que le véhicule est en mouvement. Les centres des cibles sont tracés en sur-impression vidéo en temps réel. On remarque que le segment les joignant est parallèle au support des pastilles. Il est alors simple d'en déduire l'orientation du véhicule. Le cadre en bas de l'image correspond à la zone d'entrée des cibles.

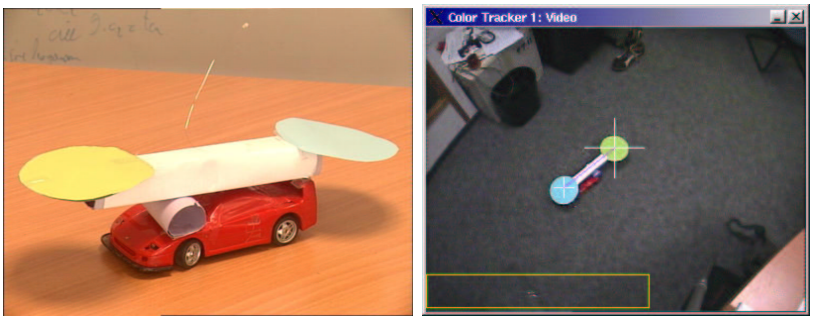


FIG. 6.11 – Suivi de pastilles pour la mesure de l'orientation d'un véhicule

Critique de la solution

Cette solution de détermination de θ est très sensible à l'instabilité numérique des deux *colortrackers*. Si celle-ci ne pose pas de problème quand il s'agit de déterminer la position d'objets dans le plan, elle est amplifiée et elle induit de fortes variations dans le calcul de θ.

La figure 6.12 présente l'histogramme de répartition des valeurs d'abscisse du centre d'une cible immobile mesurée pendant une minute (1500 mesures).

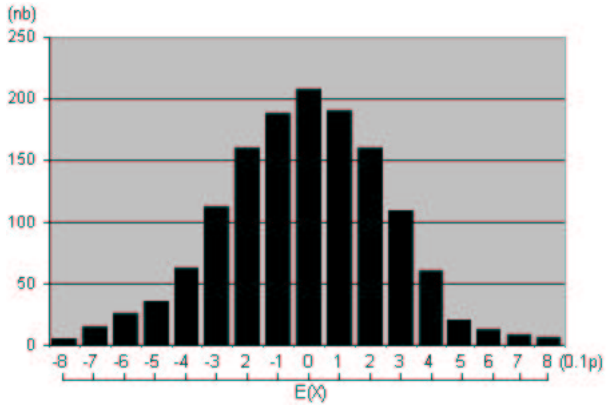


FIG. 6.12 – Histogramme de répartition des valeurs d'abscisse du centre d'une cible immobile

Cet intervalle mesuré de 1.6 pixel représente 0.41% d'erreur par rapport aux 388 pixels de définition horizontale des images vidéo utilisées. Même si cela paraît négligeable, il faut prendre en compte les effets de la perspective. Dans le cas le plus défavorable, le *Cycab* évolue à l'autre bout du parking. En prenant en compte les dimensions du parking, voir 4.1.3, la longueur focale f de nos objectifs, la largeur l et la définition def de notre capteur, il est possible de calculer l'incertitude projetée i_{proj} sur le plan des pastilles parallèle au sol à partir de notre incertitude en pixels i_{pix}. La figure 6.13 illustre cette projection.

Soit :

$$\alpha_g = \tan^{-1}\left(\frac{d}{h}\right)$$

Taille de l'incertitude sur le capteur :

$$i_c = \frac{i_{pix}.l}{def}$$

Distance angulaire de l'incertitude :

$$\alpha_i = \tan^{-1}\left(\frac{l}{2.f}\right) - \tan^{-1}\left(\frac{l - 2i_c}{2.f}\right)$$

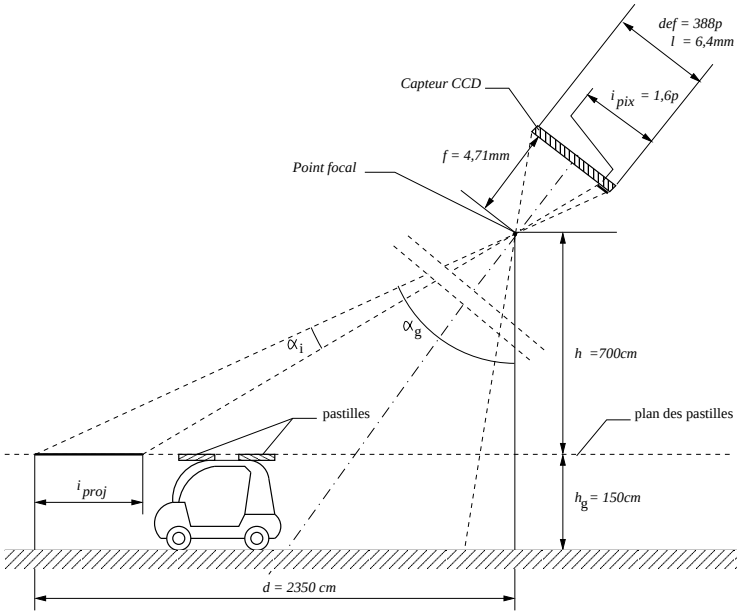


FIG. 6.13 – Projection de l'incertitude

Taille de l'incertitude sur le plan des pastilles :

$$i_{proj} = d - h.\tan(\alpha_g - \alpha_i)$$

Dans notre cas :

$$i_{proj} = 32,61cm$$

Dans le cas le plus favorable, lorsque l'axe passant par les centres des pastilles est parallèle à l'axe des abscisses de notre capteur, minimisant alors les effets de la perspective, sa variation angulaire α_{va} est donc comprise entre (voir figure 6.14) :

$$\alpha_{va} = 2.\tan\left(\frac{i_{proj}}{d}\right).$$

Dans notre cas, avec un axe de 150cm, $\alpha_{va} = 24.81°$. A 10 m de distance, un obstacle a alors une incertitude de positionnement de 4,63 mètres. Celle-ci ne permet pas d'estimer avec suffisamment de précision sa vitesse et sa trajectoire.

Cette solution n'est donc pas envisageable dans la version actuelle de *ParkView*, c'est-à-dire, en n'utilisant que deux caméras placées en hauteur sur le toit de la halle robotique. Néanmoins, il faudra la reconsidérer dans les versions ultérieures. A ce moment, le réseau de caméras couvrant le parking sera plus dense. Avec des distances moins importantes et la redondance des zones observées, il sera alors possible de minimiser les effets de l'instabilité

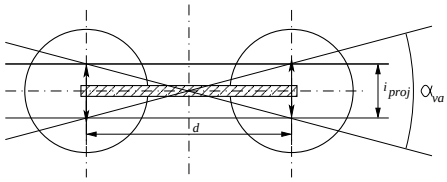


FIG. 6.14 – Variation angulaire de l'axe des pastilles

numérique des *colortrackers*.

6.2.3 Détermination de la *pose* par localisation par capteur laser

Si dans notre cas, l'utilisation de la vision n'est pas possible pour déterminer θ, l'équipe *CyberMove* a développé un système de localisation de robot mobile utilisant un capteur laser embarqué. Cette localisation utilise un ensemble de balises réfléchissantes placées aléatoirement sur le parking. La figure 6.15 présente le *Cycab* équipé du capteur laser *Sick* avec ses balises.

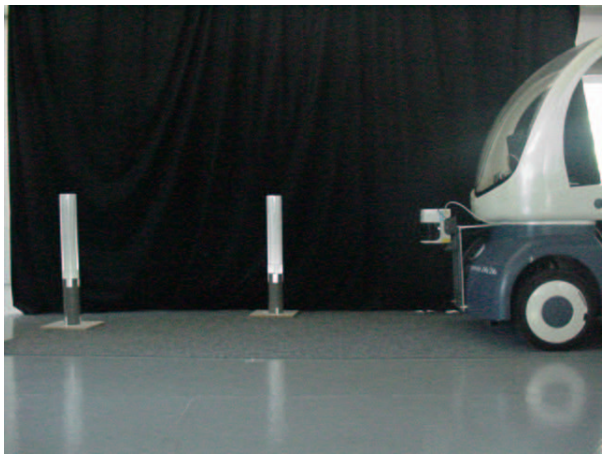


FIG. 6.15 – Bornes réfléchissantes et Capteur *Sick* monté sur le *Cycab*

Principe de la localisation par laser

L'objectif de cette partie est d'introduire l'algorithme de localisation par capteur laser Sick développé pour le *Cycab*. Elle porte sur le module de localisation présenté dans [PS02] en tant que partie du processus de localisation et de construction de carte simultanée ($SLAM$).
L'environnement dans lequel évolue le *Cycab* contient un ensemble de balises détectables par le capteur *Sick*. Celles-ci sont des cylindres recouverts d'une feuille réfléchissante permettant de les différencier des obstacles par l'intensité du faisceau renvoyé. A chaque instant, le capteur

laser perçoit donc un sous-ensemble de ces balises comme un ensemble d'observations $O = \{o_i\}$ qu'il faut identifier aux balises connues $L = \{l_j\}$, voir figure 6.16.

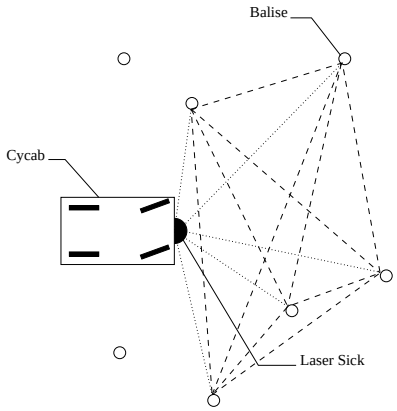


FIG. 6.16 – Localisation du *Cycab* par laser *Sick*

L'idée de cet algorithme est de comparer des caractéristiques des balises invariantes par changement de *pose* du véhicule. Par exemple la distance entre deux balises ne change pas avec la position ou l'orientation du Cycab. Il compare l'ensemble des caractéristiques extraites des observations avec celles issues d'une base de données pré-construite à partir de la connaissance des balises. De cette comparaison va donc résulter un ensemble d'hypothèses cohérentes entre elles dont on va finalement extraire l'identification finale des observations.

Graphe des hypothèses Formellement un graphe de correspondance est construit dont chaque noeud représente une hypothèse $o_i \leftrightarrow l_j$ et chaque arc exprime la cohérence entre deux hypothèses.

Soit l'algorithme :

1. Construction de la base de données de référence des caractéristiques à partir de L. Par exemple, si la caractéristique choisie est la distance entre deux balises, cette base contiendra l'ensemble des distances séparant les balises.
2. Recherche dans cette base d'une caractéristique extraite des observations et ajout des arcs entre les noeuds correspondants. Avec l'exemple de la distance comme caractéristique invariante, on recherchera pour chaque paire d'observations (o_m, o_n), la paire de balise (l_i, l_j) dont la distance se rapproche le plus de $d(o_m, o_n)$ et on ajoutera les arcs $(o_m, l_i) \leftrightarrow (o_n, l_j)$ et $(o_m, l_j) \leftrightarrow (o_n, l_i)$.
3. Recherche de la clique[1] de taille maximale dans le graphe de correspondance. La figure 6.17 présente un exemple de graphe pour trois balises et leur observation. On va

[1] Une clique est un graphe où chaque noeud est relié à tous les autres par une arête.

donc obtenir l'ensemble des hypothèses cohérentes qui vont donc définir notre mise en correspondance.

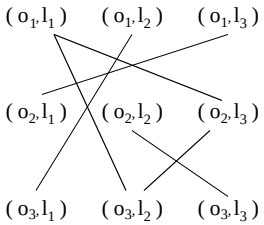


FIG. 6.17 – Graphe de correspondance entre hypothèses

A chaque nouvel ensemble d'observations, on pourra alors ré-effectuer les étapes 2 et 3 pour obtenir une nouvelle mise en correspondance entre les observations et les balises dont la position est connue.

Choix de la caractéristique invariante. Le choix de la caractéristique évaluée est responsable de la robustesse et de la rapidité du processus. La distance entre deux balises, l'angle entre trois d'entre elles sont deux invariants utilisables. Cependant il faut la choisir de manière à obtenir le moins d'ambiguïté possible. Dans le cas de la distance il faut éviter d'avoir deux paires de balises séparées par des distances trop proches pour la précision du capteur laser. De plus les angles ne sont ni faciles ni rapides à calculer.

Les caractéristiques utilisées dans [PS02] sont des triangles définis par trois balises ou observations. L'aire et la configuration permettent à la fois une robustesse de la comparaison et un faible coût calculatoire grâce à une comparaison en deux temps (aire puis configuration).

Estimation de la position. Une fois établie la correspondance entre les observations et les balises, on infère la pose du robot en minimisant :

$$F(x, y, \theta) = \sum_i \|R_\theta(o_i) + T_{x,y} - l_i\|^2$$

ou R_θ est la rotation de centre (x, y) et d'angle θ et $T_{x,y}$ la translation $(x, y)^T$, voir figure 6.18.

Cette minimisation possède heureusement une expression analytique donnée dans [PS02].

6.2.4 Prise en compte de la localisation laser dans *ParkView*

Le référentiel dans lequel travaille la localisation par laser a pour origine la position du *Cycab* lorsque celui-ci constitue sa base de distance. Il nous faut donc mettre en correspondance le repère laser et celui de référence de *ParkView*. Pour cela, le repère laser est considéré

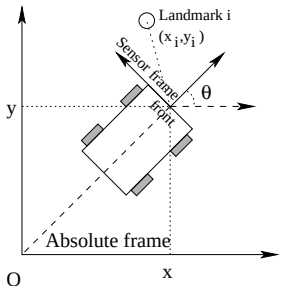


FIG. 6.18 – *Pose* du robot *Cycab*

comme un plan qui nous permet de déterminer une homographie vers le plan de référence, voir figure 6.19. Ces paramètres sont à déterminer à chaque fois que la localisation par laser est reconfigurée.

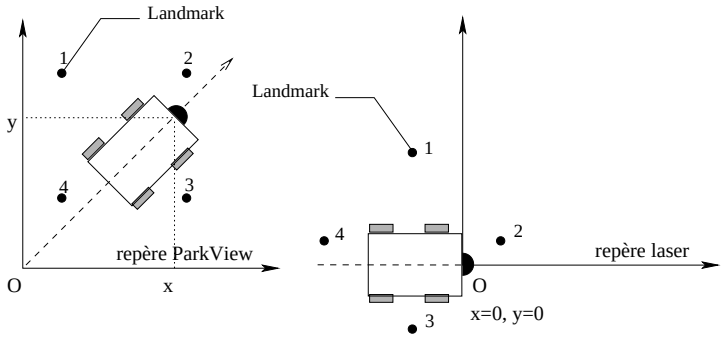


FIG. 6.19 – Repères de *ParkView* et du laser

Adaptation de `ParkviewApp`

`ParkviewApp` est modifié pour prendre en compte un nouveau type de plan. Cela se fait en créant le type `P_Laser` comme une classe dérivée de `P_Plan` (voir figure 6.20).

Modifications de l'interface graphique `Parkview_UI`

Plan vidéo L'interface graphique a été modifiée au niveau de la boîte de dialogue des instances de *landmark* dans un plan vidéo. Il est maintenant possible de saisir manuellement les coordonnées affichées par le *tracker*.

Une évolution ultérieure de l'interface devra permettre de pouvoir pointer directement la cible correspondant au *Cycab* dans l'image vidéo. Le passage des paramètres pourra se faire dans un premier temps par mémoire partagée. Lorsque l'architecture *ParkView* sera décentralisée, il se fera par le réseau.

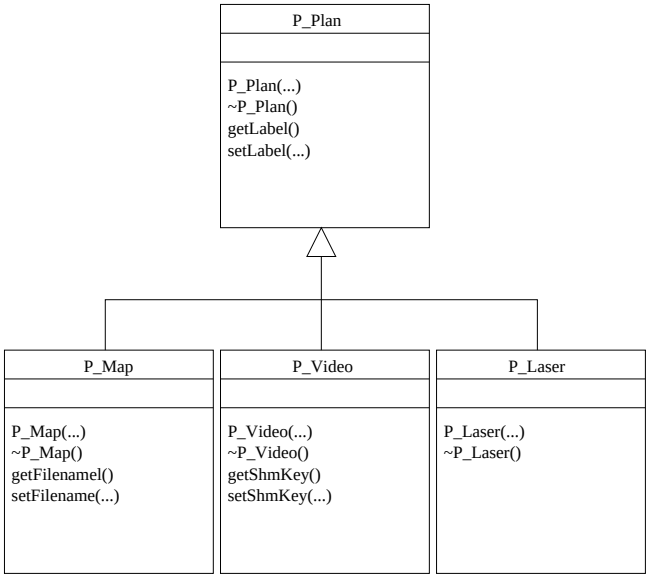


FIG. 6.20 – Spécialisation de `P_Plan` vers `P_Laser`

Plan laser Les coordonnées de la position du *Cycab* dans le plan laser sont saisies au clavier. Pour cela, la fenêtre de gestion des instances de *landmarks* a été ajoutée, voir figure 6.21.

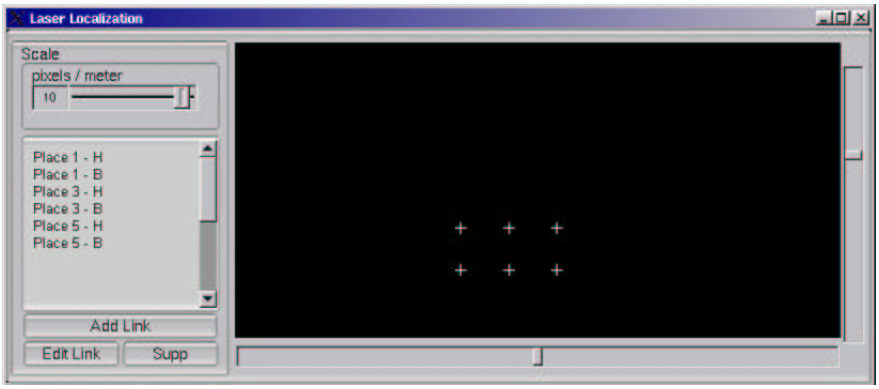


FIG. 6.21 – Fenêtre de gestion des instances de *landmark* du plan laser

Une évolution de l'interface prendra en compte les valeurs retournées par le module de positionnement laser directement. Ce module et *ParkView* fonctionnant dès à présent sur des machines différentes, la transmission devra se faire par le réseau.

6.2.5 Fusion des localisations *ParkView* et laser

Discrimination du *Cycab* parmi les cibles suivies

L'application *ParkView* renvoie indifféremment des informations concernant les obstacles mobiles et le *Cycab*. Aucune discrimination n'est faîte puisque le *tracker* n'est pas capable de distinguer le *Cycab* dans l'image. Il nous faut pour cela comparer les coordonnées du *Cycab* renvoyées par la localisation par laser, dans le repère de référence, et les objets de *ParkView*. Cela se fait de la même manière que la fusion des données provenant des deux *trackers* dont l'algorithme est développé en 3.4.2. Celui-ci est simplifié puisqu'une seule cible est gérée par la localisation laser. Il suffit alors de distinguer l'objet associé au positionnement par laser des autres.

Le module de fusion

Dans cette expérimentation, les modules de planification d'évitement d'obstacles et de localisation par laser sont embarqués à bord du *Cycab*. Son orientation, θ, n'étant pas utile aux autres modules, la fusion des données est donc effectuée par un module également embarqué utilisant l'interface `ParkviewClient`. La figure 6.22 présente l'architecture de l'application.

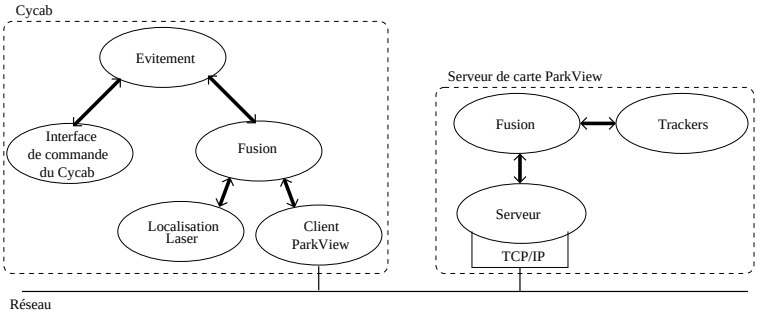


FIG. 6.22 – Architecture de l'application d'évitement d'obstacles

6.2.6 Protocoles expérimentaux

Pour l'utilisation de *ParkView* de le cadre de l'évitement d'obstacles, deux protocoles expérimentaux sont proposés. Le premier est utilisé lorsqu'on prend en compte des obstacles fixes sur le parking. Dans ce cas, ils sont référencés dans un fichier. Il faut donc travailler avec un repère de référence absolue qui ne change pas d'une expérimentation à l'autre.

Le second protocole est utilisé lorsque l'on ne tient pas compte des obstacles fixes. Il est préférable qu'il n'y en ait également pas dans la réalité.

Protocole expérimental travaillant dans le repère de *ParkView*

Ce protocole utilise le *Cycab* comme cible pour le pointage dans *landmark*.

1. Lancement du module de localisation *Sick* qui crée un repère de référence dont l'origine est la première position du *Cycab*.
2. En pilotant manuellement le *Cycab*, on vient le placer sur un *landmark* du parking.
3. Dans tous les plans vidéo où le *tracker* a détecté le *Cycab,* on associe les coordonnées des centres des cibles représentant le *Cycab* au *landmark*.
4. Dans le plan Laser, on associe la position du *Cycab* au *landmark*.
5. On répète les étapes 2, 3 et 4 pour qu'il y est un minimum de quatre *landmark* communs entre les plans vidéo et le repère de référence.
6. On génère l'ensemble des homographies vers le plan de référence.

On notera que ce protocole est fastidieux. Il sera possible, par la suite, d'automatiser une partie de ces étapes. On pourra modifier le *tracker* pour qu'il discrimine le *Cycab* des autres cibles. Cela pourrait être fait par reconnaissance d'une particularité du *Cycab* telle qu'une lampe clignotante à une fréquence connue ou encore un gyrophare. L'étape 3 serait alors facilitée. On pourrait penser également à automatiser l'étape 2, en utilisant l'un des planificateurs de trajectoire du *Cycab* développé par *CyberMove*. A ce moment, dès que la correspondance entre le repère laser et celui de référence serait connue, le *Cycab* pourrait aller seul se placer à l'aplomb de chaque *Landmark*. Une partie des itérations 2, 3 et 4 serait automatisée.

Protocole expérimental travaillant dans le repère du Sick

Le protocole présenté ici utilise une version de *ParkView* où le plan de référence est celui du capteur laser. Ceci demande des modifications de `ParkviewApp` qui n'ont pas été implémentées. Elle présente l'intérêt de ne pas devoir placer le *Cycab* à l'aplomb de *landmarks* sur le parking. Le centre du repère de référence correspondra donc à la position initiale du capteur *Sick* et il ne sera pas possible d'utiliser de fichier de coordonnées d'obstacles fixes.

1. Lancement du module de localisation *Sick*. Il crée un repère de référence dont l'origine est la première position du *Cycab*.
2. En pilotant manuellement le *Cycab*, on vient le placer dans le champ des caméras.
3. Dans tous les plans vidéo où le *tracker* a détecté le *Cycab,* on associe les coordonnées des centres des cibles représentant le *Cycab* à un même *landmark*.
4. Dans le plan de référence Laser, on associe la position du *Cycab* au *landmark*.

5. On répète les étapes 2, 3 et 4 pour qu'il y est un minimum de quatre *landmark* communs entre les plans vidéo et le repère laser de référence.
6. On génère l'ensemble des homographies.

Ici, en utilisant un *tracker* capable de discriminer le *Cycab*, il serait possible d'automatiser l'ensemble des étapes puisqu'aucune position de *landmark* dans le plan de référence doit être connue a priori.

Chapitre 7

Conclusion

Effectué dans le cadre de travail stimulant qu'est l'INRIA Rhône-Alpes, ce stage a traité de sujets aussi divers que la robotique mobile, la programmation système et client / serveur, l'optique ou encore la vision par ordinateur. C'est dans l'organisation et la réalisation de ces tâches hétérogènes qu'il trouve son intérêt.

Le travail réalisé

Nous avons ainsi mis en place *ParkView*, une architecture expérimentale ouverte, modulaire, facilement extensible, dont l'application de génération de cartes dynamiques permet à des robots mobiles d'être renseignés sur l'environnement dans lequel ils évoluent. Pour cela, en plus de nos propres développements, nous avons intégré les technologies de plusieurs équipes de recherche intéressées par notre plate-forme.

La polyvalence de *ParkView* a été mise à contribution lors du développement de deux applications concernant le mesure automatisée de la vitesse sur route et l'évitement d'obstacles mobile pour un robots.

Les développements futurs

Au cours du développement de cette première version de *ParkView*, beaucoup d'idées d'évolution sont apparues.

Il sera ainsi intéressant d'équiper l'ensemble du parking de l'*INRIA* en utilisant un système décentralisé de suivi de cible. Il faudra adapter l'algorithme de fusion afin de prendre en compte plus de deux caméras. Il sera également intéressant de pouvoir caractériser la nature des objets suivis (voiture, piéton, vélo, ...) afin de pouvoir mieux anticiper leur trajectoire.

En ce sens, l'*INRIA*, dans le cadre d'un programme de recherche commun avec le *CNRS*,

m'a proposé de participer aux futurs développements de *ParkView* dans la continuité de ce stage.

Chapitre 8

Annexes

Annexe A - Velocity circles

Le programme *Velocity Circles* permet de créer rapidement des séquences *MPEG* destinées au test des programmes de suivi de cibles. Ce programme a été utilisé pour mesurer la précision des positions et des vitesses retournées par le *tracker*. Il génère pour cela des disques de couleurs décrivant des trajectoires circulaires dans le plan image.

Le test à consisté à comparer les données retournées par le *tracker* avec les paramètres de notre programme.

Il faut noter que ce n'est pas une application. En ce sens, notre programme nécessite une recompilation à chaque modification de ses paramètres.

8.0.7 Paramètres

L'ensemble des paramètres sont fixés par l'intermédiaire de la commande `DEFINE` de pré-processeur du langage *C*.

Type de trajectoire

La définition de `CIRCLE` ou `LINE` définit le type de trajectoire à créer. Il faut noter que seule la trajectoire `CIRCLE` en `DOUBLE` permet de créer les obstacles bleus dont la définition des

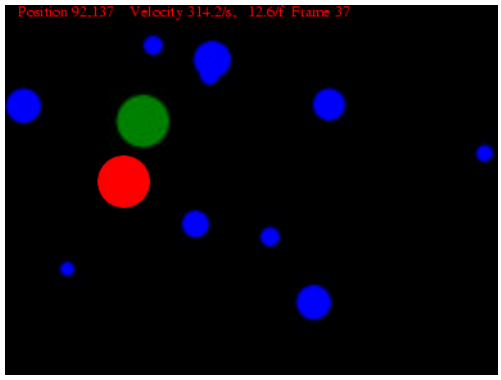


FIG. 8.1 – *Velocity Circles*

trajectoires circulaires est aléatoire.

```
// Type of trajectory  CIRCLE or LINE
#define CIRCLE
```

Nombre de cibles

Afin de simuler les deux cibles dont nous avions pensé nous servir pour déterminer l'orientation de notre robot, il est possible de générer deux disques de couleurs différentes qui décrivent la même trajectoire. Cette double cible est obtenue en définissant la constante DOUBLE. Le décalage entre les deux cibles est défini en nombre de *frames* (images) par la constante de pré-processeur `RETARD`.

```
// If double targets
#define DOUBLE
#define  RETARD    4.0   // nb frames between two targets
```

Paramètres du groupe de cibles principales

La cible principale est représentée par un ou deux disques de couleurs différentes. Il est possible de contrôler les paramètres de la trajectoire de ce groupe cible principale par la commande de pré-processeur `LINE` ou `CIRCLE`.

```
// Global parameters
#define  NB_FRAME  250    // nb frames in sequence
#define  R_DISK    20     // Radial length of disk
#define  RETARD    4.0    // nb frames between two targets

// Param for the circle trajectory
#define  TR_P_SEC  0.5    // round per seconde
#define  C_CIRC_X  192    // Center of the circle trajectiry X
#define  C_CIRC_Y  144    // Y
#define  R_CIRC_X  100.0  // Coef X of disk trajectory
#define  R_CIRC_Y  100.0  //  ... Y ...
#define  INCREAS_R 0.0    // coef. of increasing radial trajectory in pix per frame.

// Param for the line trajectory
#define O_X 0             // X coord. of start point of line trajectory
#define O_Y 144           // Y ...
```

```
#define T_X 384          // X coord. of and point of line trajectory
#define T_Y 144          // Y ...
```

Obstacles aléatoires

Afin de simuler des obstacles mobiles dans l'image, il est possible, en définissant la commande de pré-processeur `OBSTACLES`, de créer un ensemble de cibles dont les tailles et les paramètres de trajectoires circulaires sont aléatoires.

```
#define OBSTACLES        // Définintion de la commande du préprocesseur
#define NB_OBST 10       // Nb of random targets.

// Bornage des valeurs
#define OBST_CX 384.0    // Coord X max of center of random Obstacles
#define OBST_CY 288.0    //   ... Y ...
#define OBST_RT 100.0    // Length max of radius of trajectory
#define OBST_R  10.0     // Length max of radius of target
#define OBST_TPS 0.5     // Nb. of round per sec. max
```

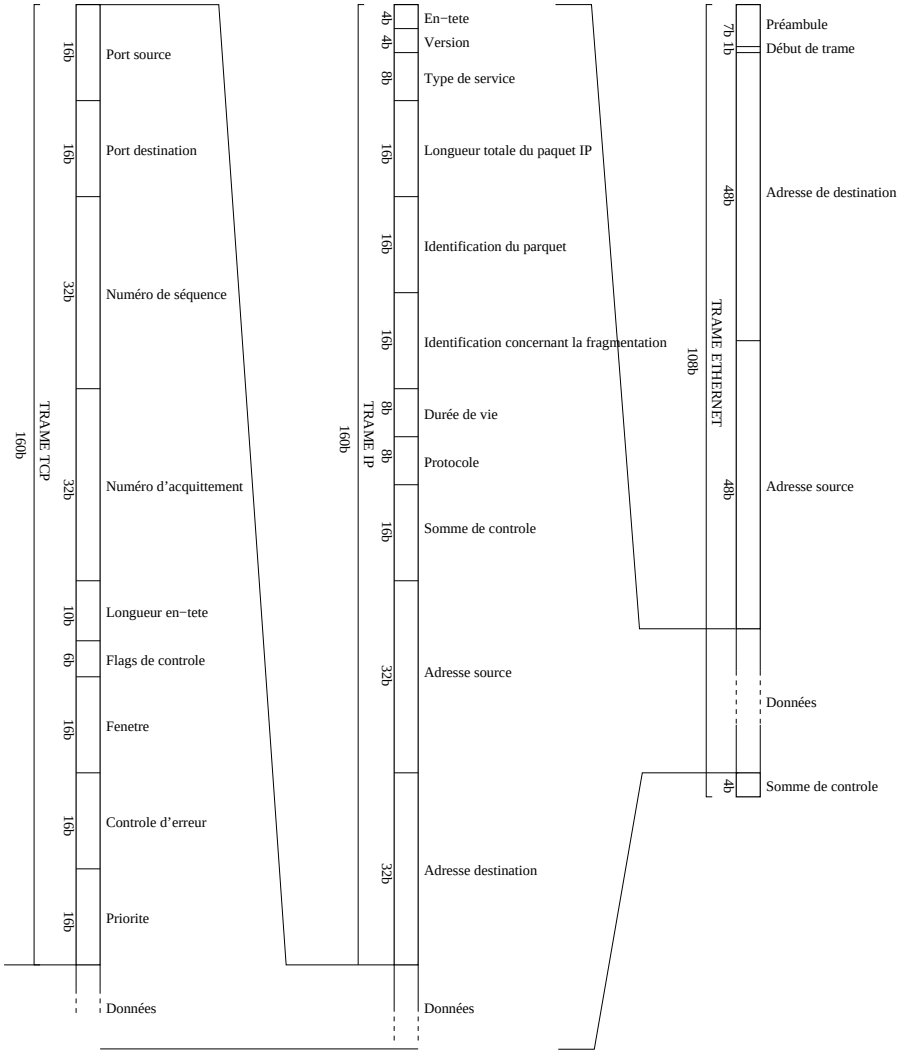
8.0.8 Lancement du programme

Le programme *VCircl* ne crée pas directement la séquence *MPEG*. Il crée un script de commandes qui utilise la programme *montage* dans sa version *ImageMagick 5.3.8 09/01/01*. Pour plus de rapidité, *VCircl* est lui-même inclu dans un script de commandes qui commande sa compilation, son lancement et la redirection de sa sortie standard vers un fichier disposant des droits d'exécution qui est à son tour exécuté. Les images fabriquées par la commande *montage* sont enregistrées dans le répertoire *images*. Finalement, la commande *mpeg-encode* est appelée par le script pour la fabrication de la séquence vidéo *MPEG* "circle.mpeg".

Script de lancement :

```
#!/bin/ksh
echo Compilation
cc -o VCircl VCircl.c -lc -lm
echo Lancement du programme
VCircl>VCircl.sh
VCircl.sh
echo Creation MPEG
mpeg_encode mpeg_encode.param
```

Annexe B - Trames Ethernet / IP /TCP



Annexe C - Test de correction de distorsion

Test de correction d'une image déformée. Nous utilisons ici la fonction de projection :

$$r_c = r_d(1 + k.r_d^2)$$

avec r_d, la distance entre le centre de distorsion et un point de l'image déformée, r_c, la distance entre le centre de distorsion et l'image de ce même point dans l'image corrigée. Le tableau 8.1 montre les effets de la transformation pour différentes valeurs de k à partir de l'image originale non corrigée présentée en figure 8.2.

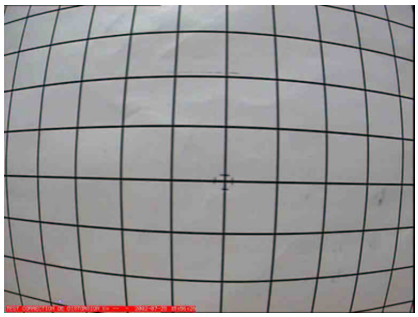
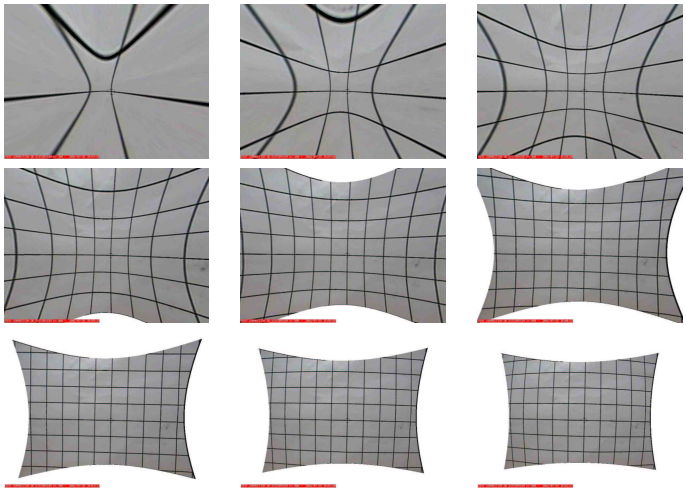


FIG. 8.2 – Image originale non corrigée



TAB. 8.1 – Visualisation du résultat de la fonction de correction de distorsion pour différentes valeurs de k.

Bibliographie

[Com00] D. Comer, *Tcp/ip architecture protocoles applications*, troisième ed., INFORMATIQUES, DUNOD, Paris, Septembre 2000.

[Cro02] James L. Crowley, *Formation et analyse d'images*, Support de cours donné à l'ENSIMAG et au DESS GI de l'Universite Joseph Fourier de GRENOBLE, 2001/2002.

[Dou00] Matthijs Douze, *Appariement d'images liées par homographie*, Master's thesis, ENS d´Électrotechnique, d´Électronique, d´Informatique, d´Hydraulique et des Télécommunications de Toulouse, septembre 2000.

[FL00] Rémy Fannader and Hervé Leroux, *Uml, principes de mise en oeuvre*, Dunod, Paris, 2000.

[Fou01] The Apache Software Foundation, *Xerces-c++ documentation*, Documentation technique, Apache Software Foundation, 2001, http ://xml.apache.org.

[FS00] Martin Fowler and Kendall Scott, *Uml distilled*, second ed., Objects Technology Series, Addison-Wesley, Massachusetts, USA, Decembre 2000.

[Ini03] Inivis, *Ac3d v3.6 user manual*, 2003.

[JVC01] JVC, *Coulour video camera tk-c1481*, Victor Compagny of Japan, Limited, Japan, 2001, Instructions Book.

[JVC02] ———, *Tk-c1480/1481/1460 command guide*, Victor Compagny of Japan, Limited, Japan, April 2002, Version 1.00.

[LSSL02a] F. Large, S. Sekhavat, Z. Shiller, and C. Laugier, *Towards real-time global motion planning in a dynamic environment using the NLVO concept*, Proc. of the IEEE-RSJ Int. Conf. on Intelligent Robots and Systems (Lausanne (CH)), September-October 2002.

[LSSL02b] ———, *Using non-linear velocity obstacles to plan motions in a dynamic environment*, Proc. of the Int. Conf. on Control, Automation, Robotics and Vision (Singapore (SG)), December 2002.

[Mar95] F. Margaret, *Perspective projection : the wrong imaging model*, 1995.

[McL99] Philip F McLauchlan, *Gandalf : The fast computer vision and numerical library*, Documentation technique, Imagineer Software Ltd., 1999, http ://gandalf-library.sourceforge.net.

[Pri01] Prima, *Rapport d'activité 2001 du projet prima-ii*, Perceiving Appearance for Robust Real-Time Computer Vision, INRIA Rhône-Alpes, Grenoble (FR), Resp. Scient : James L. Crowley, 2001.

[PS02] C. Pradalier and S. Sekhavat, *Concurrent matching, localization and map building using invariant features*, Proc. of the IEEE-RSJ Int. Conf. on Intelligent Robots and Systems (Lausanne (CH)), September-October 2002.

[PWH97] T. Pajdla, T. Werner, and V. Hlavac, *Correcting radial lens distortion without knowledge of 3-d structure technical report tr*, 1997.

[SES02] Michael Sweet, P. Earls, and Bill Spitzak, *Fltk 1.1.0 programming manual*, 2002, http ://www.fltk.org.

[Sha01] Sharp, *Rapport d'activité 2001 du projet sharp*, Programmation automatique et système décisionnel en robotique , INRIA Rhône-Alpes, Grenoble (FR), Resp. Scient : Dr. C. Laugier, 2001.

[Ste97] G. P. Stein, *Lens distortion calibration using point correspondences*, cvpr97, 1997.

[Str97] Bjarne Stroustrup, *The c++ programming language (3th ed.)*, Addison-Wesley Longman Publishing Co., Inc., 1997.

[TYO02] Toru Tamaki, Tsuyoshi Yamamura, and Noboru Ohnishi, *Correcting distortion of image by image registration*, Proc. of ACCV2002, 2002, pp. 521–526.

[WSB02] Marta Wilczkowiak, Peter Sturm, and Edmond Boyer, *Calibrage et reconstruction à l'aide de parallélépipèdes et parallélogrammes*, Actes du 13ème Congrès Francophone de Reconnaissance des Formes et Intelligence Artificielle, janvier 2002, pp. 849–857.

NOM : HELIN
Prénom : Frédéric Pierre
Sujet : Développement de la plate-forme expérimentale *ParkView* pour la reconstruction de l'environnement dynamique

Mémoire d'ingénieur C.N.A.M., Grenoble
le 1er juillet 2003

L'infrastructure *ParkView*, installée sur le parking arrière de l'*INRIA Rhône-Alpes*, est une plate-forme d'expérimentation polyvalente. Elle permet de faciliter les manipulations dans les domaines de la vision et de la robotique mobile.

Objectifs de *ParkView* Nous définissons les objectifs de *ParkView* et apportons des solutions concernant son extensibilité, l'ouverture de son architecture, la prise en compte de plusieurs capteurs et la mise à disposition de l'information en temps réel. Nous définissons également la notion de carte de l'environnement dynamique comme étant une carte remise à jour périodiquement de l'ensemble des objets mobiles présents sur le parking.

Construction de la carte de l'environnement dynamique Dans une première partie, nous présentons le dispositif de suivi de cibles utilisé. Après avoir défini le repère de référence *ParkView*, nous nous intéressons aux transformations des données issues des capteurs vers celui-ci. Nous détaillons ensuite la fusion de ces données et exposons l'algorithme utilisé pour la construction des cartes. Dans une dernière partie, nous nous intéressons aux aspects temps réel du transport des cartes et des données de suivi.

Infrastructure matérielle Nous présentons les choix faits lors de l'installation de l'infrastructure sur le bâtiment de l'*INRIA*. Ils concernent les matériels, l'étude des caractéristiques des objectifs des caméras ainsi que la conception de leur support.

Développements logiciels Après avoir exposé l'architecture générale retenue, nous décrivons l'interface graphique réalisée. Nous détaillons ensuite les structures des données de paramétrage définies. Nous présentons enfin l'architecture du serveur de cartes ainsi qu'un test de fusion de données.

Applications Nous présentons deux applications utilisant tout ou partie de *ParkView*. La première concerne la mesure automatique de la vitesse sur route. La deuxième s'intéresse à l'orientation d'un robot dans le cadre d'une expérimentation sur l'évitement d'obstacles mobiles.

Mots clés : Fusion de Données - Suivi de Cible - Transformations Homographiques - Correction de la Distorsion Optique - Programmation Asynchrone - Robot Mobile - Mesure de la Vitesse Routière.

Keywords : Data Merging - Target tracking - Transformations using homography - Correction of Optical Distortion - Asynchronous Programming - Mobile Robot - Car Velocity Measurement.